**ANSWERS — WORK OUT P.5Q BEFORE READING THIS PAGE**

# *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 ?  "Hello World"
```
1
2

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."
```
3
4

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 ?  "swap llams"
```
5
6

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

## ANSWERS — WORK OUT P.6Q BEFORE READING THIS PAGE

> ***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 ?* "tooh ot toh oot" 10
*¿ HASKELL COMMAND ?* reverse "batman & robin" 11
*HASKELL RESPONSE* • "nibor & namtab" 12
*¿ HASKELL COMMAND ?* reverse "ABLE WAS I ERE I SAW ELBA" 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 ?* True 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" 17
HASKELL RESPONSE • False 18
HASKELL COMMAND • "WAS" == reverse "SAW" 19
HASKELL RESPONSE • True 20
HASKELL COMMAND • "charlie horse" == "Charlie horse" 21
HASKELL RESPONSE • False 22
HASKELL COMMAND • "watch for spaces  " == "watch for spaces" 23
HASKELL RESPONSE • False 24
HASKELL COMMAND • "count    spaces" == "count spaces" 25
HASKELL RESPONSE • False 26

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" 35
HASKELL RESPONSE • False 36

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* ? False 34
*HASKELL COMMAND* • "ERE" == "ERE" 27
¿ *HASKELL RESPONSE* ? True 28
*HASKELL COMMAND* • "Chicago" == reverse "ogacihc" 29
¿ *HASKELL RESPONSE* ? False 30
*HASKELL COMMAND* • "Chicago" == reverse "ogacihC" 31
¿ *HASKELL RESPONSE* ? True 32

## ANSWERS — WORK OUT P.9Q BEFORE READING THIS PAGE

**_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 ← *correct answer*

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

3 And this one?
*HASKELL COMMAND •* "toh oot" == (reverse "too hot")
a True ← *correct answer*
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 ← *correct answer*

**ANSWERS — WORK OUT P.10Q BEFORE READING THIS PAGE**

# *Definitions* 2

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 ?* "ABLE WAS I ERE I SAW ELBA" 7

*HASKELL COMMAND* • reverse notPalindrome 8

*¿ HASKELL RESPONSE ?* "OGACIHC WAS I ERE I SAW ELBA" 9

*HASKELL COMMAND* • longPalindrome == reverse longPalindrome 10

*¿ HASKELL RESPONSE ?* True 11

*HASKELL COMMAND* • notPalindrome == reverse notPalindrome 12

*¿ HASKELL RESPONSE ?* False 13

*HASKELL COMMAND* • longPalindrome == shortPalindrome 14

*¿ HASKELL RESPONSE ?* False 15

*HASKELL COMMAND* • reverse squashedPalindrome == squashedPalindrome 16

*¿ HASKELL RESPONSE ?* True 17

HASKELL COMMAND • "ABLE WAS I ERE I SAW ELBA" == spacedPalindrome 18
¿ HASKELL RESPONSE ? False 19
¿ HASKELL DEFINITION ? defineThisName = "Get this response." 20
HASKELL COMMAND • defineThisName 21
HASKELL RESPONSE • "Get this response." 22

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) 23

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" 24
HASKELL RESPONSE • True 25
HASKELL COMMAND • isPalindrome "CHICAGO" 26
HASKELL RESPONSE • False 27
HASKELL COMMAND • isPalindrome longPalindrome 28
HASKELL RESPONSE • True 29

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 ?* True 32
*HASKELL COMMAND •* isPalindrome notPalindrome 33
*¿ HASKELL RESPONSE ?* False 34
*HASKELL COMMAND •* isPalindrome squashedPalindrome 35
*¿ HASKELL RESPONSE ?* True 36
*HASKELL COMMAND •* isPalindrome (reverse shortPalindrome) 37
*¿ HASKELL RESPONSE ?* True 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.

## *Review Questions*

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" ← *correct answer*
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 ← *correct answer*
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 ← *correct answer*
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.

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.

> **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.

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 ?* ABLE WAS I ERE I SAW ELBA 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!" 1

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

*HASKELL COMMAND •* longPalindrome 2

*¿ HASKELL RESPONSE ?* A man, a plan, a canal. Panama! 3

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.

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:

> ***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.

*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 ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*

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 ← *correct answer*
- 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"
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
```

*there is a space between these apostrophes*

```
HASKELL COMMAND •      removeBlanks napolean                                  4
¿ HASKELL RESPONSE ?   "AblewasIereIsawElba."                                 5
HASKELL COMMAND •      removeBlanks "s p a c e d o u t"                       6
¿ HASKELL RESPONSE ?   "spacedout"                                            7
HASKELL COMMAND •      removeBlanks maddog                                    8
¿ HASKELL RESPONSE ?   "Hegoddammaddog,eh?"                                   9
HASKELL COMMAND •      removeBlanks hot                                       10
¿ HASKELL RESPONSE ?   "toohottohoot"                                         11
HASKELL COMMAND •      removeBlanks(reverse chicago)                          12
¿ HASKELL RESPONSE ?   ".ogacihCwasIereIsawelbA"                              13
HASKELL COMMAND •      removeBlanks hot == reverse(removeBlanks hot)          14
¿ HASKELL RESPONSE ?   True                                                   15
¿ HASKELL DEFINITION ?                 -- function to remove periods (you write it)
¿ HASKELL DEFINITION ? removePeriods str =
¿ HASKELL DEFINITION ?    [c | c <- str, c /= '.']                            16
HASKELL COMMAND •      removePeriods chicago                                  17
HASKELL RESPONSE •     "Able was I ere I saw Chicago"                         18
HASKELL COMMAND •      removeBlanks(removePeriods chicago)                    19
HASKELL RESPONSE •     "AblewasIereIsawChicago"                               20
¿ HASKELL DEFINITION ?       -- function to remove blanks and periods (you write it)
¿ HASKELL DEFINITION ? removeBlanksAndPeriods str =
¿ HASKELL DEFINITION ?    removeBlanks(removePeriods str)                     21
HASKELL COMMAND •      removeBlanksAndPeriods napolean                        22
HASKELL RESPONSE •     "AblewasIereIsawElba"                                  23
```

## *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 ← *correct answer*
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 ← *correct answer*

***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 ← *correct answer*

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 ]` ← *correct answer*

*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 ← *correct answer*

# *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." 1a
*HASKELL COMMAND •* removePeriods(removeBlanks madam) 1

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 /= ','] 2a
*HASKELL COMMAND •* removeCommas(removePeriods(removeBlanks madam)) 2

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 3

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 ?* (removeCommas . removePeriods . removeBlanks) str 4
*HASKELL COMMAND •* removeBPC madam 5.c1
*HASKELL RESPONSE •* "MadamImAdam" 6.c1

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 ?          [c | c <- str, c /= chr]                             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] 19

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 7.c2

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 ' ' 18

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.

## *Review Questions*

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 ← *correct answer*
c syntax error, type conflict in operands
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

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 ← *correct answer*
c syntax error, type conflict in operands
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

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 ← *correct answer*
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

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 ← *correct answer*
d xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

5 If equals is a function that requires two arguments, then equals 'x' is a function that requires

a no arguments
b one argument ← *correct answer*
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."   1
HASKELL RESPONSE •  "MadamI'mAdam"                                               2
```

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.

## ANSWERS — WORK OUT P.27Q BEFORE READING THIS PAGE

***operators as arguments***

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

*function composition as an argument*

*function composition as an operator*

```
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 ' '
```
3a.fc18

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

*¿ HASKELL DEFINITION ?* `removeBPC = foldr1 (.) [remove ',', remove '.', remove ' ']` 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`
>
> *argument type*
>
> *double-colon reads "has 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." ]` *mapping* toLower *onto the* 4
*HASKELL RESPONSE •* `"madam, i'm adam."` *sequence* "Madam, I'm Adam." 5

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 ? [toLower c | c <- str] 6
HASKELL COMMAND • allLowerCase "Madam, I'm Adam." 6a
HASKELL RESPONSE • "madam, i'm adam." 6b

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 <- ",. "] 7

*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 <- ",. "] 8

*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 ? foldr1 (.) [remove c | c <- ",. ;:'\"?!()"] 9
HASKELL COMMAND • removePunctuation "Madam, I'm Adam." 10
HASKELL RESPONSE • "MadamImAdam" 11

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 ?     isPalindrome . removePunctuation . allLowerCase          12
HASKELL COMMAND •        isPalindromic "Able was I ere I saw Elba."                 13
HASKELL RESPONSE •       True                                                       14
HASKELL COMMAND •        isPalindromic "Able was I ere I saw Chicago."              15
HASKELL RESPONSE •       False                                                      16
```

> ***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***

Your answer to the preceding question may have taken a form that did not use the function composition operator, or it may have used the operator only once instead of twice.

Either of the following definitions would also be correct.

```
HASKELL DEFINITION •   isPalindromic phrase =
HASKELL DEFINITION •       isPalindrome((removePunctuation . allLowerCase) phrase)
HASKELL DEFINITION •
HASKELL DEFINITION •   isPalindromic phrase =
HASKELL DEFINITION •       isPalindrome((removePunctuation(allLowerCase phrase)))
```

However, it is worthwhile taking the time to figure out why the form using two composition operations is correct. This will give you some insight into the use of this operation.

## *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 ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*

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

5 The second argument of foldr1 must be
- a a sequence ← *correct answer*
- 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" ← *correct answer*
  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 ← *correct answer*
  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.

***type-inquiry command***

HASKELL COMMAND • reverse "abcdefg" 1
HASKELL RESPONSE • "gfedcba" 2
HASKELL COMMAND • :type reverse "abcdefg" 3

:type *command inquires about the type of a formula*

*formula about whose type the* :type *command is inquiring*

*formula whose type follows the double-colon*

*double-colon reads “has type”*

*type delivered by formula*

HASKELL RESPONSE • reverse "abcdefg" :: [Char] 4

[Char] *is a synonym of* String

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

## ANSWERS — WORK OUT P.35Q BEFORE READING THIS PAGE

HASKELL COMMAND • :type ["aardvark", "boar", "cheeta"] 9
¿ HASKELL RESPONSE ? ["aardvark", "boar", "cheeta"] :: [ [Char] ] 10
HASKELL COMMAND • reverse ["aardvark", "boar", "cheeta"] 13
¿ HASKELL RESPONSE ? ["cheeta", "boar", "aardvark"] 14
HASKELL COMMAND • :type reverse ["aardvark", "boar", "cheeta"] 11
¿ HASKELL RESPONSE ? reverse ["aardvark", "boar", "cheeta"] :: [ [Char] ] 12

### *Review Questions*

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

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

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

4 The formula foldr1 (.) [f, g, h, k] delivers
- a a string
- b a function ← *correct answer*
- 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 ← *correct answer*

6 The type of the formula foldr1 (.) [f, g, h, k] is
- a String -> String
- b the same as the type of f ← *correct answer*
- 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] ← *correct answer*
- d none of the above

## ANSWERS — WORK OUT P.36Q BEFORE READING THIS PAGE

8 The Haskell entity [ [ "Beyond", "Rangoon" ], [ "Belle", "du", "Jour" ] ] has type

- a [ [ String ] ]
- b [ [ [ Char ] ] ]
- c both of the above ← *correct answer*
- 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 15
*HASKELL RESPONSE* • reverse :: [a] -> [a] 16

> ***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 20
*HASKELL COMMAND* • :type isPalindromic 21
*HASKELL RESPONSE* • isPalindromic :: [Char] -> Bool 22

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] 23
*HASKELL COMMAND* • :type allLowerCase 24
*HASKELL RESPONSE* • allLowerCase :: [Char] -> [Char] 25

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 26
*¿ HASKELL RESPONSE ?* toLower :: Char -> Char 27

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) 17.d23
*HASKELL COMMAND* • :type isPalindrome 18
*HASKELL RESPONSE* • isPalindrome :: Eq a => [a] -> Bool 19

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.

# ANSWERS — WORK OUT P.39Q BEFORE READING THIS PAGE

*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]     28
HASKELL COMMAND •     :type remove                                  30
HASKELL RESPONSE •    remove :: Eq a => a -> [a] -> [a]             29
```

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***

*function whose type is being declared*

*types of arguments*

*first* *second*

*type of value delivered by function*

```
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 ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*

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 ← *correct answer*

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 ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*

# ANSWERS — WORK OUT P.41Q BEFORE READING THIS PAGE

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 ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*
- 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 ? remove '?' :: [Char] -> [Char] 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***

f :: a -> b ← f *transforms arguments of type* a *to values of type* b
x :: a ← x *has type* a
*implies*
f x :: b ← f x *has type* b

g :: a -> b -> c ← g *transforms two arguments (of types* a *and* b*, resp.) to a value of type* c
x :: a ← x *has type* a
*implies*
g x :: b -> c ← g x *transforms one argument (of type* b*) to a value of type* c

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 removePunctuation?

HASKELL COMMAND • :type [remove c | c <- " ,.;:'\"?!()" ] 37
¿ HASKELL RESPONSE ? [remove c | c <- " ,.;:'\"?!()" ] :: [ [Char] -> [Char] ] 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 (.)` 39

*¿ HASKELL RESPONSE ?* `(.) :: (a -> b) -> (c -> a) -> (c -> b)` 40

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** :: (a -> a -> a) -> [a] -> a
>
> 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 ?* foldr1 :: (a -> a -> a) -> [a] -> a 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] ← *correct answer*
   - 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 ← *correct answer*
   - 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]] ← *correct answer*
   - 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) ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*

7 A function whose type is `(a -> b) -> c` must be
- a lower order
- b middle order
- c higher order ← *correct answer*
- 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 ← *correct answer*
- 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

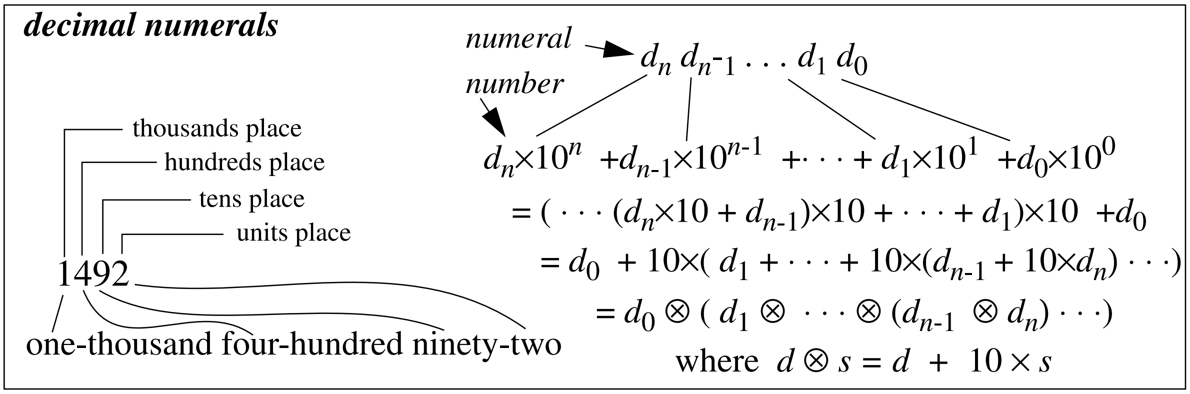
The examples of the next few chapters have to do with different ways to represent information, from numbers to encrypted text. **Information representation** is another central theme of computer science. For this reason, the examples themselves are as important to your education as the programming methods 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 $d_n\, d_{n\text{-}1} \ldots d_1\, d_0$, where $d_i$ is the digit in the $10^i$ position, then the number the numeral denotes is the following sum:

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

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 $10^{n\text{-}1}$, 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:

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

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.

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

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 [$d_0$, $d_1$, …, $d_{n-1}$, $d_n$] 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 ?     foldr1 multAdd ds                                  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 ?      foldr1 multAdd ds
¿ HASKELL DEFINITION ?      where
¿ HASKELL DEFINITION ?      multAdd  d  s = d + 10*s
```
4

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 ?       (horner10 . reverse) ds                               5
  HASKELL COMMAND •       integerFromDecimalNumeral [1,4,9,2]                      6
  HASKELL RESPONSE •      1492                                                     7
```

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** :: (a -> b -> b) -> b -> [a] -> b
>
> 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 ?* 70

*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 ? foldr multAdd 0 ds
¿ HASKELL DEFINITION ? where
¿ HASKELL DEFINITION ? multAdd d s = d + 10*s 8
HASKELL COMMAND • horner10 [1, 2, 3, 4]
HASKELL RESPONSE • 4321
HASKELL COMMAND • horner10 [ ]
HASKELL RESPONSE • 0
HASKELL COMMAND • integerFromDecimalNumeral [ ]
HASKELL RESPONSE • 0

## *Review Questions*

1 Integral numbers are denoted in Haskell by
- a decimal numerals enclosed in quotation marks
- b decimal numerals — no quotation marks involved ← *correct answer*
- 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 ← *correct answer*

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 ← *correct answer*

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 ← *correct answer*
- 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 ← *correct answer*
- 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 ← *correct answer*

7 What value does the following command deliver?

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

## ANSWERS — WORK OUT P.53Q BEFORE READING THIS PAGE

*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))) ← *correct answer*
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 ?  5                                  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 ?  8                                  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      7
HASKELL COMMAND •     hundredsDigit 1517                                             8
¿ HASKELL RESPONSE ?  5                                                              9
```

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) ← *correct answer*
  - 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 ← *correct answer*
  - 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 ← *correct answer*
  - 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) ← *correct answer*

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 ← *correct answer*

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 ← *correct answer*
  - c surplus components on either side of the equation are ignored
  - d all of the above

# *The Class of Numbers* 12

The function horner10 is polymorphic. It operates on a class of numeric types.

HASKELL DEFINITION • horner10 :: Num a => [a] -> a

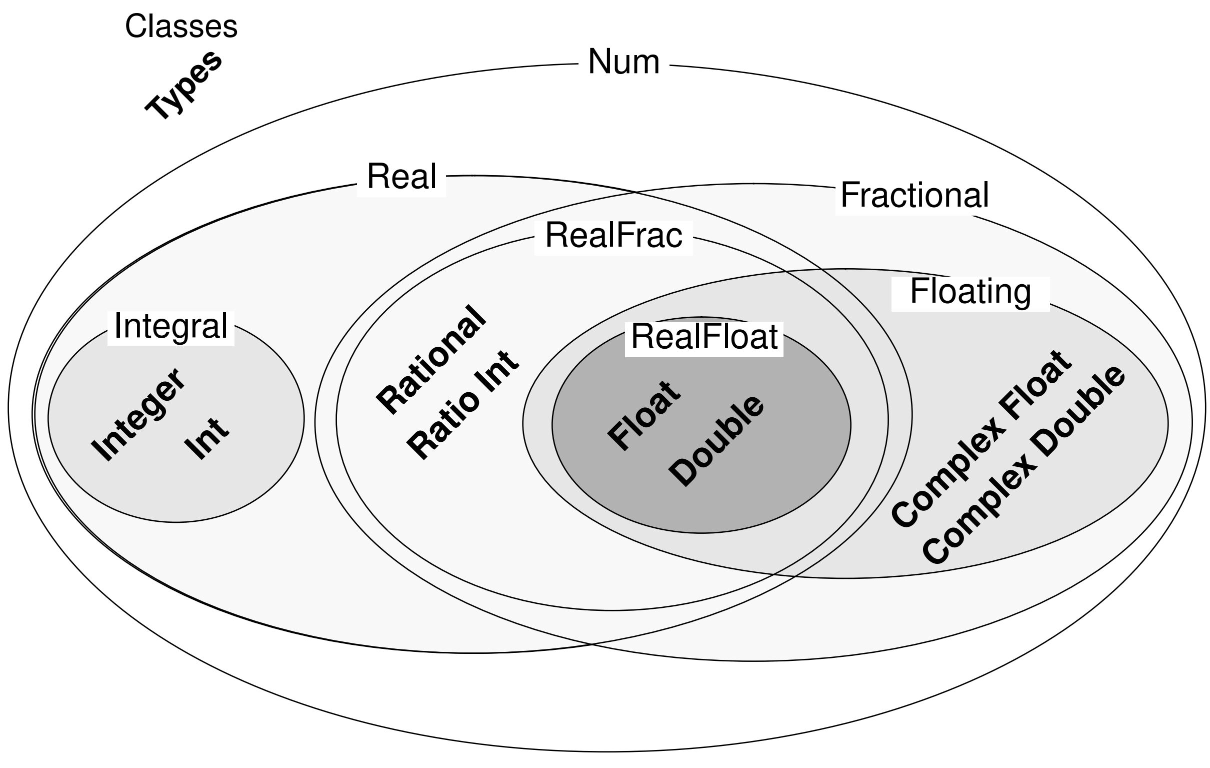
This type specification says that the argument of horner10 does not have to be Integral. It can be of any type belonging to the class Num.

Num is a class containing a total of six subclasses and eight specific types. So far the only specific type from class Num that you have seen is Integer. The type Integer is one of two types in the class Integral. The class Integral is a subclass of the class Real, and the class Real is, in turn, one of the two primary subclasses of the class of numbers, which is called Num.

One way to view the class structure of Num is to look at it as a Venn diagram. In the diagram, a region that is wholly contained in another region indicates a subclass relationship. Overlapping regions represent classes that share some of their types and subclasses. Specific types that belong to a particular class are displayed inside the region representing that class.

Each class of numbers shares a collection of operations and functions. For example, a value from any of the eight types in the class Num is a suitable operand for addition (+), subtraction (-), or multiplication (∗), and a suitable argument for negation (negate), absolute value (abs), signum (signum), or conversion from Integer to another numeric type (fromInteger). These are the seven intrinsic operations shared among all types in class Num.

***The Class of Numbers***



Other numeric operations are restricted to subclasses. For example, ‘mod‘ and ‘div‘ require operands from the class Integral, which means their operands must either be of type Integer or of type Int (integers restricted to the range[1] $-2^{29}$ to $2^{29}$-1).

Another example is division (/). Operands of the division operator must be in the class Fractional. Some of the types in the class Fractional are represented in what is known as floating point form. Floating point numbers have a fixed number of digits, but a decimal point that can be shifted over a wide range to represent large numbers or small fractions. On most computer systems, the type Float carries about seven decimal digits of precision, and Double carries about sixteen digits.

***Why several types of numbers?***

Primarily to make efficient computation possible. The instructions sets of most computers provide instructions to do fast arithmetic on three types: Int, Float, and Double. In theory, the type Integer would be adequate for convenient programming of any Integral computation. The type Int wouldn’t be needed. In practice, on the other hand, operations on numbers of type Integer proceed at a pace that could be a hundred times slower than computations with numbers of type Int. Sometimes, you just don’t have a hundred times longer to wait. That’s what Int is for, to make the computation go faster when you don’t need extra range.

There are many other functions and operators associated with various subclasses of Num. You can learn about them on an as needed basis, referring to the *Haskell Report* where necessary.

In this chapter, the only new class of numbers you need to know about is Integral. As you can see in the diagram this includes two types: Int and Integer. Both types are denoted in Haskell by decimal numerals, prefixed by a minus sign (-) in case they are negative. The difference between the two types is that one has a restricted range and the other has an unlimited range.

The official Haskell requirement is that any integer in the range $-2^{29}$ to $2^{29}$-1 is a legitimate value of type Int. Outside that range, there are no guarantees.

Some of the intrinsic functions in Haskell that will be needed in the next chapter deal with values of type Int. These intrinsic functions convert between values of type Char and values of type Int. Because of the way Haskell represents characters, there are only 255 different values of type Char. So, the type Int has plenty of range to handle integer equivalents of values of type Char, and the designers of Haskell didn’t see much point in doing a complicated conversion when a simple one would do.

Given the information that addition (+) and multiplication (∗) can operate on any type in the class Num and that ’divMod’ must have operands from the class Integral, try to figure out the most general possible types of the following functions.

```
¿ HASKELL DEFINITION ?    horner :: Num num => num -> [num] -> num
¿ HASKELL DEFINITION ?    horner b ds = foldr  (multAdd b)  0  ds
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    multAdd :: Num num => num -> num -> num -> num
¿ HASKELL DEFINITION ?    multAdd b  d  s =  d + b*s
```

1. This range is required of all Haskell systems. Usually the range will depend on the underlying hardware. For example, $-2^{31}$ to $2^{31}$-1 is the range of integers supported by hardware arithmetic on many popular chip architectures, so that is the range of values of type Int on most Haskell systems.

## ANSWERS — WORK OUT P.59Q BEFORE READING THIS PAGE

```
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    integerFromNumeral :: Num num => num -> [num] -> num
¿ HASKELL DEFINITION ?    integerFromNumeral b x = (horner b  .  reverse) x
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    numeralFromInteger :: Integral num => num -> num -> [num]
¿ HASKELL DEFINITION ?    numeralFromInteger b x =
¿ HASKELL DEFINITION ?       reverse [d | (s,d) <- takeWhile (/= (0,0)) (sdPairs b  x)]
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    sdPairs :: Integral num => num -> num -> [(num, num)]
¿ HASKELL DEFINITION ?    sdPairs b x  = iterate (nextDigit b) (x `divMod` b)
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    nextDigit :: Integral num => num -> (num, any) -> (num, num)
¿ HASKELL DEFINITION ?    nextDigit b (xShifted, d) = xShifted `divMod` b                1
```

*Hint on a tough one:* nextDigit ignores the second component in the tuple supplied as its second argument, so it doesn't care what type that component has.

### *Review Questions*

1 In the Haskell class of numbers, Int and Integer
- a are basically the same type
- b are the same type except that numbers of type Integer can be up to 100 digits long
- c are different types but x+y is ok, even if x is of type Int and y is of type Integer
- d are different types, but both in the Integral subclass
- e ← *correct answer*

2 In the Haskell class of numbers, Float and Double
- a are basically the same type
- b are the same type except that numbers of type Double can be up to 100 digits long
- c are different types but x+y is ok, even if x is of type Float and y is of type Double
- d are different types, but both in the RealFrac subclass ← *correct answer*
- e

3 What is the most restrictive class containing both the type Integer and the type Float?
- a Num
- b Real ← *correct answer*
- c RealFrac
- d Fractional

4 In the Haskell formula n/d, the numerator and denominator must be in the class
- a Integral
- b RealFrac
- c Fractional ← *correct answer*
- d Floating

# ANSWERS — WORK OUT P.60Q BEFORE READING THIS PAGE

5 What is the type of the function f?

*HASKELL DEFINITION* • f x y = x / y

a Float -> Float -> Float

b Real num => num -> num -> num

c Fractional num => num -> num -> num ← *correct answer*

d Floating num => num -> num -> num

6 What is the type of the formula ( g n 1) ?

*HASKELL DEFINITION* • g x y = x + y

*HASKELL DEFINITION* • n :: Int

*HASKELL COMMAND* • g n 1

a Int ← *correct answer*

b Integer

c Integral

d Real

# *Iteration and the Common Patterns of Repetition* 13

Look again at the definition of the function hundredsDigit from the previous chapter:

```
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     1
```

All of the definitions in the where-clause perform the same operation on different data, and the data flows from one definition to the next. That is, information generated in the first definition is used in the second, and information generated in the second definition is used in the third. It is as if a function were applied to some data, then the same function applied again to the result produced in the first application, and finally the same function applied a third time to the result produced in the second application. This illustrates a common programming method known as iteration.

> ***iteration***
>
> To iterate is to do the same thing again and again. In software, this amounts to a succession of applications of the same function, repeatedly, to the result of the previous application. In other words, to form a composition with several applications of the same function:
>
> (f . f) x — *2 iterations of* f
> (f . f . f . f . f) x — *5 iterations of* f

Technically, in software, iteration requires composing a function with itself. First, you apply the function to an argument. That's one iteration. Then, you apply the function again, this time to the result of the first iteration. That's another iteration. And so on.

The where-clause in the definition of the function hundredsDigit almost meets this technical definition of iteration, but not quite. The missing technicality is that, while data generated in one iteration is used in the next, it is not used in exactly the form in which it was delivered.

The first iteration delivers the tuple (xSansLastDigit, d0), and the second iteration uses only the first component of this tuple to deliver the next tuple (xSansLast2Digits, d1). The third iteration follow the same practice: it uses on the first component of the tuple to compute the third tuple. With a little thought, one can iron out this wrinkle and define hundredsDigit in the form of true iteration in the technical sense.

The trick is to define a function that generates the next tuple from the previous one. This function will ignore some of the information in its argument:

```
HASKELL DEFINITION •   nextDigit(xShifted, d) = xShifted `divMod` 10     2
   HASKELL COMMAND •   nextDigit(151, 7)     3
¿ HASKELL RESPONSE ?   (15, 1)     4
```

The nextDigit function can be used to define hundredsDigit in a new way, using true iteration:

```
¿ HASKELL DEFINITION ?   hundredsDigit  x  = d2
¿ HASKELL DEFINITION ?      where
¿ HASKELL DEFINITION ?      (xSansLast3Digits, d2) = (nextDigit . nextDigit) (x `divMod` 10)     5
```

This scheme leads to a simple formula for extracting any particular digit from a number: put together an *n*-stage composition of nextDigit to extract digit *n* of a decimal numeral, where *n* represents the power of ten for which that digit is the coefficient:

HASKELL COMMAND • x ‘divMod‘ 10 — *extracts digit* 0
HASKELL COMMAND • nextDigit (x ‘divMod‘ 10)— *extracts digit* 1
HASKELL COMMAND • (nextDigit . nextDigit) (x ‘divMod‘ 10)— *extracts digit* 2
HASKELL COMMAND • (nextDigit . nextDigit . nextDigit) (x ‘divMod‘ 10)— *extracts digit* 3

The above formulas are iterations based on the function nextDigit. Each formula delivers a two-tuple whose second component is the extracted digit. This formulation of digit extraction suggests a way to derive a complete decimal numeral from a number: just build the sequence of digits through successively longer iterations of the function nextDigit:

```
[d0, d1, d2, d3, …] = [ d | (s, d) <- [ x ‘divMod‘ 10,
                                        nextDigit (x ‘divMod‘ 10),
                                        (nextDigit . nextDigit) (x ‘divMod‘ 10),
                                        (nextDigit . nextDigit . nextDigit)  (x ‘divMod‘ 10),
                                           …   ]
                      ]
```

*tuple-patterns can be used in generators, too*

The Haskell language provides an intrinsic function to build the sequence of iterations described in the above equation. The function is called iterate, and its two arguments are a (1) a function to be repeated in the iterations and (2) an argument for that function to provide a starting point for the iterations.

For example, if iterate were applied to the function that adds one to its argument and to a starting point of zero, what sequence would it generate?

HASKELL DEFINITION • add1 n = n + 1
HASKELL COMMAND • iterate add1 0
¿ HASKELL RESPONSE ? [0, 1, 2, 3, 4, 5, … ]

***Hint***

```
iterate add1 0 =
  [0, add1 0, (add1 . add1) 0,
     (add1 . add1 . add1) 0,
       (add1 . add1 . add1 . add1) 0,
         …  ]
```

In a similar way, an invocation of iterate can generate the powers of two. In this case, instead of adding one, the iterated function doubles its argument.

HASKELL DEFINITION • double p = 2∗p
HASKELL COMMAND • iterate double 1
HASKELL RESPONSE • [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, … ]

Combining these two ideas leads to a formula for the sequence of tuples in which the first component is the power to which the base (2) is raised and the second component is the corresponding power of two.

HASKELL DEFINITION • add1Double (n, p) = (n + 1, 2∗p)
HASKELL COMMAND • iterate add1Double (0, 1)
HASKELL RESPONSE • [(0, 1), (1, 2), (2, 4), (3, 8), (4, 16), (5, 32), … ]

## ANSWERS — WORK OUT P.63Q BEFORE READING THIS PAGE

In general, the function iterate builds a sequence that reflects the cumulative effects of applying a given function, repeatedly, to an argument supplied as a starting point.

**iterate** :: (a -> a) -> a -> [a]

iterate $f$ $x$ = $[d_0, d_1, d_2, d_3, \ldots]$
where
$[d_0, d_1, d_2, d_3, \ldots] = [\, x,\ f\,x,\ (f\,.\,f)\,x,\ (f\,.\,f\,.\,f)\,x,\ \ldots\,]$

- iterate generates a sequence with an infinite number of elements
- calculations that use iterate will truncate the sequence it generates when the elements needed in the computation are delivered

*The above definition of iterate is intended to describe the result that* iterate *delivers. The definition uses some Haskell syntax, but is not written purely in Haskell. The following equation defines* iterate *formally in Haskell. Don't try to puzzle out its meaning at this point — you don't have all the necessary information yet.*

*HASKELL DEFINITION •* `iterate f x = [x] ++ iterate f (f x)` 6

Now take another look at the formulas for the digits of the decimal numeral of x:

```
[ x ‘divMod‘ 10,
   nextDigit (x ‘divMod‘ 10),
      (nextDigit . nextDigit) (x ‘divMod‘ 10),
         (nextDigit . nextDigit . nextDigit)  (x ‘divMod‘ 10),   ...   ]
```

Try to use the function iterate to write a Haskell function that delivers the sequence of digits in the decimal numeral of a given integer. Don't worry, at this point, about the sequence being infinite.

```
¿ HASKELL DEFINITION ?   allDigitsInNumeralStartingFromUnitsPlace  x =
¿ HASKELL DEFINITION ?     [d | (s,d) <- sdPairs x]
¿ HASKELL DEFINITION ?   sdPairs x  =  iterate nextDigit  (x `divMod` 10)
```
7

```
HASKELL COMMAND •   allDigitsInNumeralStartingFromUnitsPlace 1863
HASKELL RESPONSE •  [3, 6, 8, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0,^C{Interrupted!}
```

*interrupted by pressing control-C — otherwise, it keeps on going like the Energizer Bunny*

***non-terminating computations and embedded software***

Iteration is one of the mechanisms that makes it possible for programmers to describe non-terminating computations in Haskell. Such computations are needed in some types of software. For example, the software that controls an ATM (automatic teller machine) describes a non-terminating computation: serving a customer amounts to one session of an unlimited number of sessions that occur as customers, one after another, use the machine. Most software embedded in devices for control purposes has this non-terminating characteristic.

Device control is an area that has been truly liberated by the advent of small, cheap computer chips. In the days of mechanical controls, most devices sensed external conditions with the same components that actuated controls. There was little opportunity for translating conditions detected by sensors into complex control sequences. Electronic controls have made it possible to do a great deal of analysis of conditions and to take different actions based on those conditions. Fuel injection systems, anti-lock braking systems, sophisticated stereo equipment, music synthesizers, electronic thermostats, and multi-function wrist watches are examples of the benefits of this technology.

**zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]**
**zipWith op** [$x_1, x_2, x_3, \ldots$] [$y_1, y_2, y_3, \ldots$] **=**
**[op** $x_1\ y_1$**, op** $x_2\ y_2$ **, op** $x_3\ y_3$ **, … ]**

*Note:* zipWith *delivers a sequence whose length matches that of the shorter of the sequences supplied as its last two arguments. Zipping stops when one of those arguments runs out of elements.*

**filter :: (a -> Bool) -> [a] -> [a]**
**filter keep xs = [x | x <- xs, keep x]**

***filtering as folding:***

```
filter keep  = foldr op [ ]
    where
    op x ys
      | keep x      = [x] ++ ys
      | otherwise   = ys
```

*result if* keep x *is* True

*if not*

**map :: (a -> b) -> [a] -> [b]**
**map f xs = [f x | x <- xs]**

***mapping as folding:***

```
map f  = foldr op [ ]
    where
    op x ys = [f x] ++ ys
```

***mapping as zipping:***

```
map f = zipWith op (repeat(error "ignored"))
    where
    op ignored x = f x
```

Iteration consists of repeating the same computation over and over. You have seen two other forms of repeated computation, mapping and folding, for which Haskell provides intrinsic operators. You have also used list comprehension to express the idea of filtering — that is, selecting certain elements from a given sequence to form a new one.

There is an intrinsic function filter for this transormation, which can be defined as a folding process. Most computations calling for repetition fall into one of these patterns or into a pattern called zipping, which is a generalized form of mapping.

When you are trying to describe a computation that involves repetition, try to view it as one of common patterns: mapping, folding, filtering, iteration, and zipping. The operators map, foldr, filter, iterate, and zipWith make up a kind of linguistic shorthand that covers probably over 90% of the computations involving repetition that you will encounter in practice.

It pays to try to view repeated computations in one of the common patterns because you will acquire a facility for quickly understanding computations specified in these ways, and this will make it more likely that your programs will do what you expect. And, other people will find it easier to understand your programs when you write them in this way.[1]

## *Review Questions*

1 The iterate function
- a delivers an infinite sequence as its value
- b applies a function to the value that function delivers, over and over
- c delivers its second argument as the first element of a sequence
- d all of the above ← *correct answer*

2 What value do the following Haskell commands deliver?

```
HASKELL DEFINITION •   add2 n = n + 2
HASKELL COMMAND •      iterate add2 0
HASKELL COMMAND •      iterate add2 1
```

- a the biggest number that Haskell can compute
- b nothing — they aren't proper commands
- c the number that is two more than the starting point
- d one delivers the sequence of even numbers, the other the odds ← *correct answer*
- e

3 Use the iterate function to generate the sequence $[x_0, x_1, x_2, x_3, \ldots]$ where $x_0 = 1$ and $x_{n+1} = 11x_n \bmod 127$.
- a `next x = x/127 * 11`
  `iterate next 1`
- b `next x = (11*x) ‘mod‘ 127`
  `iterate next (1/11 ‘div‘ 127)`
- c `next x = (11*x) ‘mod‘ 127`
  `iterate next 1` ← *correct answer*
- d none of the above

> ***pseudorandom numbers***
> Sequences like $[x_0, x_1, x_2, x_3, \ldots]$ (in which each successive element is the remainder, using a fixed divisor, when the previous element is multiplied by a fixed multiplier) sometimes exhibit many of the statistical properties of random sequences. This is the usual way of generating "random" numbers on computers.

1. There is another reason for using standard operators to specify repetition: efficiency. People who develop systems that carry out Haskell programs realize that most repeating computations will be described in a standard way, so they invest a great deal of effort to ensure that their systems will use computing resources efficiently when performing one of the common repetition operations.

# *Truncating Sequences and Lazy Evaluation* 14

Sometimes a computation will need to work with part of a sequence. To accommodate situations like these, Haskell provides intrinsic functions that accept a sequence as an argument and deliver part of the sequence as a result. Four such functions are take, drop, takeWhile, and dropWhile.

The function take delivers an initial segment of a sequence. Its first argument says how many elements to include in the initial segment to be delivered, and its second argument is the sequence whose initial segment is to be extracted. The function drop delivers the other part of the sequence — that is, the sequence without a specified number of its beginning elements.

**take, drop** :: Int -> [a] -> [a]

take $n$ $[x_1, x_2, \ldots, x_n, x_{n+1}, \ldots] = [x_1, x_2, \ldots, x_n]$
drop $n$ $[x_1, x_2, \ldots, x_n, x_{n+1}, \ldots] = [x_n, x_{n+1}, \ldots]$

*HASKELL COMMAND* • take 3 [1, 2, 3, 4, 5, 6, 7]
*¿ HASKELL RESPONSE ?* [1, 2, 3]
*HASKELL COMMAND* • drop 3 [1, 2, 3, 4, 5, 6, 7]
*¿ HASKELL RESPONSE ?* [4, 5, 6, 7]
*HASKELL COMMAND* • (take 3 . drop 2) [1, 2, 3, 4, 5, 6, 7]
*¿ HASKELL RESPONSE ?* [3, 4, 5]
*HASKELL COMMAND* • take 3 [1, 2]
*HASKELL RESPONSE* • [1, 2] — *takes as many as are available*
*HASKELL COMMAND* • drop 3 [1, 2]
*HASKELL RESPONSE* • [ ] — *drops as many as it can; delivers empty list*

Try to define a function that delivers a sequence of the thousands digit through the units digit of the decimal numeral denoting a given number. Use the function take and the function allDigitsInNumeralStartingFromUnitsPlace (defined in the previous chapter) in your definition.

*¿ HASKELL DEFINITION ?* lastFourDecimalDigits x = -- you define it
*¿ HASKELL DEFINITION ?* (reverse . take 4 . allDigitsInNumeralStartingFromUnitsPlace) x
*HASKELL COMMAND* • lastFourDecimalDigits 1937
*HASKELL RESPONSE* • [1, 9, 3, 7]
*HASKELL COMMAND* • lastFourDecimalDigits 486
*¿ HASKELL RESPONSE ?* [0, 4, 8, 6]
*HASKELL COMMAND* • lastFourDecimalDigits 68009
*¿ HASKELL RESPONSE ?* [8, 0, 0, 9]

The functions takeWhile and dropWhile are similar to take and drop, except that instead of truncating sequences based on counting off a particular number of elements, takeWhile and dropWhile look for elements meeting a condition specified in the first argument.

The first argument of takeWhile and dropWhile is a function that delivers Boolean values (True/False). This function is called a **predicate**. As long as elements in the sequence pass the test specified by the predicate (that is, as long as the predicate delivers True when applied to an element from the initial part of the list), takeWhile continues to incorporate these elements into the sequence it delivers. When takeWhile encounters an element that fails to pass the test, that element and all that follow it in the sequence are truncated (actually, they are never generated in the first place — see box on lazy evaluation).

***lazy evaluation***

The Haskell system always waits until the last minute to do computations. Nothing is computed that is not needed to deliver the next character of the result demanded by the command that initiated the computation in the first place.

So, when takeWhile is applied to a sequence, the only elements of the sequence that will ever be generated are those up to and including the first one that fails to pass takeWhile's test of acceptance (that is, its predicate).

This is known as lazy evaluation, and it has many consequences of great value in software design.

The function dropWhile delivers the elements from the trailing portion of the list that takeWhile would truncate: take and takeWhile truncate a trailing segment of a sequence, and drop and dropWhile truncate an initial segment of a sequence..

**takeWhile, dropWhile** :: (a -> Bool) -> [a] -> [a]

takeWhile $p$ [$x_1$, $x_2$, …] = [$x_1$, $x_2$, …$x_{k-1}$ ]
dropWhile $p$ [$x_1$, $x_2$, …] = [$x_k$, $x_{k+1}$, …]

*where $x_k$ is the first element such that $p\ x_k$ is* False

*HASKELL COMMAND* • takeWhile odd [3, 1, 4, 1, 5, 9, 2, 6]
*¿ HASKELL RESPONSE ?* [3, 1]

odd *is an intrinsic function that delivers* True *if its argument is not divisible by two*

*HASKELL COMMAND* • dropWhile odd [3, 1, 4, 1, 5, 9, 2, 6]
*¿ HASKELL RESPONSE ?* [4, 1, 5, 9, 2, 6]

***operator section***
- curried form of the less-than function (<)
- (< 5) x *is equivalent to* x < 5

*HASKELL COMMAND* • takeWhile (< 5) [3, 1, 4, 1, 5, 9, 2, 6]
*HASKELL RESPONSE* • [3, 1, 4, 1]
*HASKELL COMMAND* • dropWhile (< 5) [3, 1, 4, 1, 5, 9, 2, 6]
*HASKELL RESPONSE* • [5, 9, 2, 6]

The takeWhile function provides the means to the goal of writing a function to build the decimal numeral of a given integer. The function allDigitsInNumeralStartingFromUnitsPlace, developed in the previous chapter, almost does the trick. But, it delivers too many digits (an infinite number) and it delivers them backwards (the units digit first, then tens digit, etc.). The function contains essentially the right ideas, but needs to incorporate some sort of truncation.

With takeWhile as the truncation mechanism, the criterion for deciding where to truncate requires some tricky analysis.

Reconsider the function sdPairs that was defined for the benefit of allDigitsInNumeralStartingFromUnitsPlace:

*HASKELL DEFINITION •* `sdPairs x = iterate nextDigit (x `divMod` 10)` 8

The sdPairs function builds the sequence of tuples that allDigitsInNumeralStartingFromUnitsPlace extracts the digits of the decimal numeral from.

*HASKELL COMMAND •* `sdPairs 1954` 9

*HASKELL RESPONSE •* [(195,4), (19,5), (1,9), (0,1), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,0), (0,^C{Interrupted!} 10

The digits in the numeral for the integer $x$ are the second-components of the tuples in the sequence sdPairs $x$. That is, the $d$-components of the following sequence are the digits of the numeral.

$$\text{sdPairs x} = [(s_0, d_0), (s_1, d_1), (s_2, d_2), \ldots]$$

$$\text{where } s_{k+1} = s_k \text{ `div` } 10 \text{ and } d_{k+1} = s_k \text{ `mod` } 10 \text{ for all } k \geq 0$$

The above formulas imply that if $s_k$ is zero, then both $s_{k+1}$ and $d_{k+1}$ are zero, which means that as soon as $s_k$ becomes zero, no subsequent elements of the sequence will contain non-zero digits for the numeral. So, to build the numeral, all elements of the sequence delivered by sdPairs $x$ beyond the first one where the $s$-component is zero can be truncated.

On the other hand, the numeral contains all non-zero digits that occur in the $d$-component, so no such components can be truncated from the sequence if the numeral is to be constructed from what is left.

The upshot of these two observations is this: to construct the numeral, all the sequence elements will be needed up to, but not including, the first element where both the $s$-component and the $d$-component are zero. So, the truncation function should take elements $(s, d)$ from the sequence as long as $(s, d) \neq$ (0,0), but it can truncate all elements starting from the first one that equals (0,0).

Obviously, lazy evaluation is going to be very important in this computation. If the Haskell system weren't lazy, it might go on forever generating more and more $(s, d)$-pairs that would never be needed.

***operator sections***

As you know, an operator with two operands becomes a functions with two arguments when its operator-symbol is enclosed in parentheses.

When one operand of the operator is supplied inside the parentheses, the resulting function has only one argument. The other argument is the value specified as an operand:

| | |
|---|---|
| (== 0) x | x == 0 |
| (< 0) x | x < 0 |
| (‘div‘ 10) x | x ‘div‘ 10 |
| (c+) x | c + x |
| (1-) x | 1 - x |

Exception! (-c) is not an operator section
It is a number — the negative of c

The predicate to test for this condition would be the operator section (/= (0,0)), so the truncation formula would be

```
takewhile (/= (0,0)) (sdPairs x)
```

Imbed this truncation in the formula defining the function allDigitsInNumeralStartingFromUnitsPlace, then apply reverse to get the digits in the conventional order (units digit last) to construct a function that builds a sequence containing the digits of the decimal numeral representing a given number:

*¿ HASKELL DEFINITION ?* decimalNumeralFromInteger x = -- you define it
*¿ HASKELL DEFINITION ?* reverse [d | (s,d) <- takeWhile (/= (0,0)) (sdPairs x)] 11
*HASKELL COMMAND •* decimalNumeralFromInteger 1975 12
*HASKELL RESPONSE •* [1, 9, 7, 5] 13

## *Review Questions*

1 What is the value of w?
*HASKELL DEFINITION •* u, v, w :: String
*HASKELL DEFINITION •* u = "Four vier cuatro"
*HASKELL DEFINITION •* v = drop 5 u
*HASKELL DEFINITION •* w = drop 5 v
a "Four "
b "vier "
c "cuatro" ← *correct answer*
d "cinco"

2 What string does the following command deliver?
*HASKELL COMMAND •* takeWhile (/= blank) "Four score and seven"
a "score and seven"
b " score and seven"
c "Four "
d "Four" ← *correct answer*

3 What string does the following command deliver?
*HASKELL COMMAND •* dropWhile (/= blank) "Four score and seven"
a "score and seven"
b " score and seven" ← *correct answer*
c "Four "
d "Four"

4 What value does the following command deliver?
*HASKELL DEFINITION •* dozen = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]
*HASKELL COMMAND •* [ take 2 xs | xs <- iterate (drop 2) dozen ]
a [1, 2, 3, 4, 5, 6]
b [ [1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [ ], [ ], [ ], [ ], [ ], … ] ← *correct answer*
c [2, 4, 6, 8, 10, 12]
d [ [1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12] ]

5 Which of the following formulas delivers the product of the numbers in the sequence xs?
a dropWhile (/= 0) (iterate (∗) xs)
b takeWhile (/= 0) (iterate (∗) xs)
c foldr (∗) 0 xs
d foldr (∗) 1 xs ← *correct answer*

6 Given the following definition, which of the formulas delivers the number 3?
*HASKELL DEFINITION •* k x y = x
*HASKELL DEFINITION •* first, second, third :: Integer

*HASKELL DEFINITION •* first = k (4-1) 0
*HASKELL DEFINITION •* second = k (1+2) "three"
*HASKELL DEFINITION •* third = k 3 (1 ‘div‘ 0)

a first
b second
c third
d all of the above ← *correct answer*

7 Consider the following function.

*HASKELL DEFINITION •* f :: String -> [String]
*HASKELL DEFINITION •* f w = [take 2 w, drop 2 w]

What does the formula iterate f "cs1323" deliver?

a [ "cs", "1323", [ ], [ ], …
b [ "cs", "13", "23", [ ], [ ], …
c [ ["cs"], ["1323"], [ ], [ ], …
d error … type mismatch ← *correct answer*

# Encapsulation — modules 15

The where-clause provides one way to hide the internal details of one software component from another. Entities defined in a where-clause are accessible only within the definition that contains the where-clause. So, the where-clause provides a way to encapsulate information within a limited context. This keeps it from affecting other definitions. But, the most important reason for using a where-clause is to record the results of a computation that depends on other variables whose scope is limited to a particular context (formal parameters of functions, for example), for use in multiple places within the definition containing the where-clause. It is best to keep where-clauses as short as possible. When they get long, they mix up the scopes of many variables, which can lead to confusion.

Access to entities can also be controlled by defining them in software units known as modules. Entities defined in modules may be public (accessible from outside the module) or private (accessible only inside the module).This makes it possible to define software units that are independent of each other, except with regard to the ways in which their public entities are referred to. This, in turn, makes it possible to improve internal details in modules without affecting other parts of the software. Private entities within a module are said to be encapsulated in the module. .

```
HASKELL DEFINITION •  module DecimalNumerals
HASKELL DEFINITION •        (integerFromDecimalNumeral,  --export list
HASKELL DEFINITION •         decimalNumeralFromInteger)
HASKELL DEFINITION •     where
HASKELL DEFINITION •
HASKELL DEFINITION •     integerFromDecimalNumeral ds = (horner10 . reverse) ds
HASKELL DEFINITION •
HASKELL DEFINITION •     decimalNumeralFromInteger x =
HASKELL DEFINITION •        reverse [d | (s,d) <- takeWhile (/= (0,0)) (sdPairs x)]
HASKELL DEFINITION •
HASKELL DEFINITION •     horner10 ds = foldr  multAdd  0  ds
HASKELL DEFINITION •
HASKELL DEFINITION •     multAdd  d  s = d + 10*s
HASKELL DEFINITION •
HASKELL DEFINITION •     sdPairs x = iterate nextDigit (x `divMod` 10)
HASKELL DEFINITION •
HASKELL DEFINITION •     nextDigit(xShifted, d) = xShifted `divMod` 10
```

*encapsulating decimal numeral functions*

*file: DecimalNumerals.hs*

1

Modern programming languages[1] provide good facilities for handling this sort of encapsulation — that is, for sharing information among a particular collection of functions, but hiding it from the outside world. Haskell provides this facility through modules.

1. Haskell, ML, Java, Fortran 90, and Ada, for example — but not C and not Pascal

## ANSWERS — WORK OUT P.72Q BEFORE READING THIS PAGE

A **module** is a script that designates some of the entities it defines as exportable to other scripts, but keeps all of its other definitions to itself. Other scripts using the module may use its exportable definitions, but they have no access to its other definitions.

The module DecimalNumerals contains definitions for functions to convert between decimal numerals and integers. The module makes the definitions of the functions integerFromDecimalNumeral and decimalNumeralFromInteger available to the outside world by designating them in the export list after the module name at the beginning of the module. The other functions defined in the module are private.

A module script begins with the keyword module, which is followed by a name for the module. The module name must start with a capital letter. After the module name comes a list of the entities that will be available to scripts using the module. This is known as the **export list**. Entities not specified in the export list remain private to the module and unavailable to other scripts.

Following the export list is a where clause in which the functions of the module are defined. The module DecimalNumerals defines the functions integerFromDecimalNumeral, decimalNumeralFromInteger, horner10, multAdd, and nextDigit, all but two of which are private to the module.

A script can import the public definitions from a module, then use them in its own definitions. The script does this by designating the module in an import specification prior to the script's own definitions. If a script has no definitions of its own, it may consist entirely of import specifications. Each import specification in a script gives the script access to some of the public entities defined in the module that the import specification designates, namely those public entities designated in the import list of the import specification.

The following script imports the two public functions of the DecimalNumerals module. When this script is loaded, the Haskell system responds to commands using either of the two public functions of DecimalNumerals designated in the import list of the import specification. But the Haskell system will not be able to carry out commands using any of the private functions in DecimalNumerals. They cannot be imported.

```
HASKELL DEFINITION •    import DecimalNumerals
HASKELL DEFINITION •       (integerFromDecimalNumeral, decimalNumeralFromInteger)      2
HASKELL COMMAND •       integerFromDecimalNumeral [1, 9, 9, 3]
¿ HASKELL RESPONSE ?    1993
HASKELL COMMAND •       decimalNumeralFromInteger 1993
¿ HASKELL RESPONSE ?    [1, 9, 9, 3]
HASKELL COMMAND •       (integerFromDecimalNumeral . decimalNumeralFromInteger) 1993
¿ HASKELL RESPONSE ?    1993
HASKELL COMMAND •       nextDigit(199, 3)
HASKELL RESPONSE •      ERROR: Undefined variable "nextDigit"
```

From this point on, most of the Haskell software discussed in this text will have a main module that acts as the basis for entering commands. This main module will import functions from other modules, and the imported functions, together with any functions defined in the main module, will be the only functions (other than instrinsic functions) that can be invoked in commands. The preceding script, which imports the public functions of the DecimalNumerals module, is an example of a “main module” of this kind.

> ***module files***
> By convention, each module is defined in a file — one module to a file — with a filename that is identical to the module name plus a .hs extension. For example, the DecimalNumerals.hs file would contain the DecimalNumerals module. Exception: the file containing the main module should be given a name indicative of the software’s purpose. Otherwise, there will be too many files called Main.hs.

The following redevelopment of the numeral conversion functions provides some practice in encapsulation and abstraction.

As you know, decimal numerals are not the only way of representing numbers. Not by a long shot! There are lots of completely unrelated notations (Roman numerals, for example), but the decimal notation is one of a collection of schemes in which each digit of a numeral represents a coefficient of a power of a **radix**

In the decimal notation, the radix is ten, but any radix will do. Most computers use a radix two representation to perform numeric calculations. People use radix sixty representations in dealing with time and angular measure.

> ***radix*** — the base of a number system
>
> $d_n d_{n-1} \ldots d_1 d_0$
>
> is a radix $b$ numeral for the number
>
> $d_n \times b^n + d_{n-1} \times b^{n-1} + \cdots + d_1 \times b^1 + d_0 \times b^0$
>
> - each $d_i$ is a radix $b$ digit.
> - radix $b$ digits come from the set $\{0, 1, \ldots b\text{-}1\}$

The functions defined in the module DecimalNumerals can be generalized to handle any radix by replacing the references to the radix 10 by a parameter. For example, the function horner10 would be replaced by a new function with an additional parameter indicating what radix to use in the exponentiations. The following module for polynomial evaluation exports the new horner function. The module also defines a multAdd function that factors in the radix (its first argument), but this function is private to the module.

```
¿ HASKELL DEFINITION ?  module PolynomialEvaluation   -- you write the export list
¿ HASKELL DEFINITION ?      (horner)
¿ HASKELL DEFINITION ?    where
¿ HASKELL DEFINITION ?    horner b ds = foldr  (multAdd b)  0  ds
¿ HASKELL DEFINITION ?    multAdd b  d  s =                   -- you write multAdd
¿ HASKELL DEFINITION ?      d + b*s
```
3

The PolynomialEvaluation module can be used to help build the following module to handle numerals of any radix. Some of the details are omitted, to give you a chance to practice.

The module Numerals imports the module PolynomialEvaluation. This makes it possible to use the function horner in within the Numerals script (but not the function multAdd, which is private to the PolynomialEvaluation module).

The Numerals module exports the functions integerFromNumeral and numeralFromInteger, which are analogous to the more specialized functions that the DecimalNumerals module exported. The module does not export any other functions, however. So, a script would not get access to the function horner by importing the module Numerals.

The difference between the functions in Numerals and those in DecimalNumerals is that the ones in Numerals have parameterized the radix. That means that the functions in Numerals have an additional argument, which specifies the radix as a particular value when the functions are invoked. You can construct the functions in Numerals by using the radix parameter in the same ways the number 10 was used in the DecimalNumerals module.

```
¿ HASKELL DEFINITION ?   module Numerals
¿ HASKELL DEFINITION ?          (integerFromNumeral,
¿ HASKELL DEFINITION ?           numeralFromInteger)
¿ HASKELL DEFINITION ?      where
¿ HASKELL DEFINITION ?      import PolynomialEvaluation(horner)
¿ HASKELL DEFINITION ?      integerFromNumeral b x =        -- your turn to define a function
¿ HASKELL DEFINITION ?          (horner b  .  reverse) x
¿ HASKELL DEFINITION ?      numeralFromInteger b x =           -- you write this one, too
¿ HASKELL DEFINITION ?          reverse [d | (s,d) <- takeWhile (/= (0,0)) (sdPairs b  x)]
¿ HASKELL DEFINITION ?      sdPairs b x  =         -- still your turn
¿ HASKELL DEFINITION ?          iterate (nextDigit b) (x `divMod` b)
¿ HASKELL DEFINITION ?      nextDigit b (xShifted, d) = xShifted `divMod` b       4
```

Once the module Numerals is defined, the functions in the module DecimalNumerals can redefined in terms of the functions with a parameterized radix, simply by specifying a radix of 10 in curried invocations of the functions that Numerals exports.

```
HASKELL DEFINITION •   module DecimalNumerals
HASKELL DEFINITION •          (integerFromDecimalNumeral,
HASKELL DEFINITION •           decimalNumeralFromInteger)
HASKELL DEFINITION •      where
HASKELL DEFINITION •      import Numerals(integerFromNumeral, numeralFromInteger)
HASKELL DEFINITION •      integerFromDecimalNumeral ds = integerFromNumeral 10 ds
HASKELL DEFINITION •      decimalNumeralFromInteger x = numeralFromInteger 10 x      5
```

A main module could now import functions from either the Numerals module or the DecimalNumerals module (or both) and be able to use those functions in commands:

```
HASKELL DEFINITION •   import Numerals(integerFromNumeral, numeralFromInteger )
HASKELL DEFINITION •   import DecimalNumerals(decimalNumeralFromInteger)      6
HASKELL COMMAND •      decimalNumeralFromInteger 2001
HASKELL RESPONSE •     [2, 0, 0, 1]
HASKELL COMMAND •      numeralFromInteger  2   2001
HASKELL RESPONSE •     [1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1]
HASKELL COMMAND •      numeralFromInteger 60 138
HASKELL RESPONSE •     [2, 18]
```

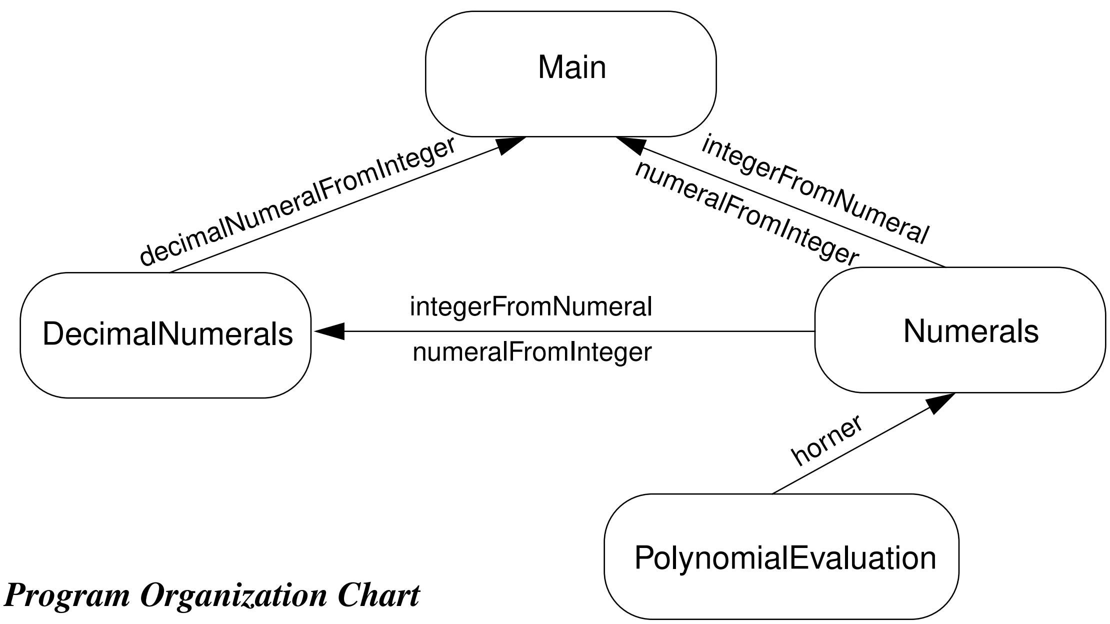
*— a hundred thirty-eight minutes is two hours, eighteen minutes*

| | | |
|---|---|---|
| HASKELL COMMAND • | numeralFromInteger 60 11697 | *— a whole bunch of seconds is* |
| HASKELL RESPONSE • | [3, 14, 57] | *three hours, fourteen minutes and fifty-seven seconds* |
| HASKELL COMMAND • | numeralFromInteger 12 68 | *— sixty-eight inches is* |
| HASKELL RESPONSE • | [5, 8] | *five feet eight inches* |
| HASKELL COMMAND • | integerFromNumeral 5280 [6, 1000] | *— six miles and a thousand feet is* |
| HASKELL RESPONSE • | 32680 | *a cruising altitude of thirty-two thousand six hundred eighty feet* |

Now you need to know how to communicate modules to Hugs, the Haskell system you have been using. Put each module in a separate file with a name identical to the name of the module, but with a ".hs" extension (or a .lhs extension if you are using the literate form in your script). When Hugs loads a Haskell script that imports a module, it finds the script defining the module by using the module's name to construct the name of the file containing the definition. So, the loading of module scripts occurs automatically, as needed.

The way in which a program is organized in terms of modules is an important aspect of its overall structure. Export lists in module specifications and import lists in import specifications reveal the details of this structure, but in a form that is scattered across files and hard to picture all at once. Another representation of the modular structure of the program, a documentation tool known (in this text, at least) as a **program organization chart**, does a better job of communicating the big picture.

A program organization chart consists of ovals linked by arrows. Each oval names a module of the program, and an arrow from one module-oval to another indicates that the module at the head of the arrow imports entities from the module at the tail. The imported entities appear on the chart as labels on the arrow.[1]




***Program Organization Chart***

1. Since the program organization chart contains no information that is not also specified in the modules, it would be best to have program organization charts drawn automatically from the definitions of the modules. This would ensure their correctness. However, the charts serve also as a good planning tool. Sketching the program organization chart before writing the program, then revising it as the program evolves helps keep the overall structure of the program in mind, which can lead to improvements in design.

## *Review Questions*

1 A Haskell module provides a way to
- a share variables and functions between scripts
- b hide some of the variables and functions that a script defines
- c package collections of variables and functions to be used in other scripts
- d all of the above ← *correct answer*

2 The export list in a module designates variables and functions that
- a are defined in the module and redefined in other modules
- b are defined in the module and will be accessible to other scripts ← *correct answer*
- c are defined in other scripts and needed in the module
- d are defined in other scripts and redefined in the module

3 An import specification in a script
- a makes all the definitions in a module available in the script
- b designates certain variables and functions in the script to be private
- c makes some public definitions from another module available for use in the script ← *correct answer*
- d specifies the importation parameters that apply in the script

4 In a numeric representation scheme based on radix $b$,
- a numbers are denoted by sequences whose elements come from a set of $b$ digits ← *correct answer*
- b numbers are written backwards
- c letters cannot be used to represent digits
- d numbers larger than $b$ cannot be represented

5 Horner's formula
- a computes the reverse of a sequence of digits
- b takes too long to compute when $n$ is bigger than 10
- c expresses a sum of multiples of powers of a certain base as a nest of products and sums ← *correct answer*
- d is too complicated to use in ordinary circumstances

# *Definitions with Alternatives* 16

Julius Caesar wrote messages in a secret code. His scheme was to replace each letter in a message with the third letter following it in the alphabet. In a coded message, he would have written URPH for ROME, for example. The following script provides functions to encode and decode messages using Caesar's cipher.

```
HASKELL DEFINITION •   cipherJulius :: String -> String
HASKELL DEFINITION •   cipherJulius = map(shiftAZ 3)
HASKELL DEFINITION •
HASKELL DEFINITION •   decipherJulius :: String -> String
HASKELL DEFINITION •   decipherJulius = map(shiftAZ (-3))
HASKELL DEFINITION •
HASKELL DEFINITION •   shiftAZ n c = ltrFromInt((intFromLtr c + n) `mod` 26)
HASKELL DEFINITION •
HASKELL DEFINITION •   intFromLtr :: Char -> Int
HASKELL DEFINITION •   intFromLtr c = fromEnum c - fromEnum 'A'
HASKELL DEFINITION •
HASKELL DEFINITION •   ltrFromInt :: Int -> Char
HASKELL DEFINITION •   ltrFromInt n = toEnum(n + fromEnum 'A')                1

HASKELL COMMAND •      cipherJulius "VENIVIDIVICI"
HASKELL RESPONSE •     "YHQLYLGLYLFL"
HASKELL COMMAND •      decipherJulius "YHQLYLGLYLFL"
HASKELL RESPONSE •     "VENIVIDIVICI"
```

You are probably wondering what the formula defining cipherJulius means. What is that map thing anyway? This is an intrinsic function that duplicates one of the uses of list comprehensions:

```
map f  xs = [f x | x <- xs]
```

It's as simple as that. So, why use map? Mainly because it makes some formulas a bit more concise. An equivalent formula for cipherJulius would be

```
cipherJulius plaintext = [shiftAZ 3  c | c <- plaintext]
```

Taking f to be the curried form shiftAZ 3 in the definition of map, this formula for cipherJulius msg equivalent to the following:

```
cipherJulius  plaintext  =  map (shiftAZ 3) plaintext
```

This definition of cipherJulius is almost the same as the original. The only difference is, this one names an explicit argument, and the original uses a curried invocation of map, leaving the argument implicit. But, the definitions are equivalent because of the following observation:

$$f\ x\ =\ g\ x \text{ for all } x \qquad \text{means} \qquad f = g$$

This is what it means for two functions to be the same: they deliver the same values when supplied with the same arguments. Because Haskell allows curried function invocations, the mathematical idea of function equality carries over to the syntax of Haskell. The following two Haskell definitions are equivalent, no matter how complicated the *anyFormula* part is:

f x = *anyFormula* x    is equivalent to    f = *anyFormula*

The same trick works if the f part is a curried form:

g y z x = *anyFormula* x    is equivalent to    g y z = *anyFormula*

So,

```
cipherJulius msg = map (shiftAZ 3) msg
```

is equivalent to

```
cipherJulius     = map (shiftAZ 3)
```

From now on, you'll see this form of expression in lots of definitions. When definitions omit some of the parameters of the function being defined, subtle ambiguities[1] can arise. For this reason, it is necessary to include explicit type specifications for such functions. Generally, explicit type specifications are good practice anyway, since they force the person making the definition to think clearly about types. So, most definitions from this point on will include explicit type specifications.

Now, back to the script for computing Caesar ciphers.

The function shiftAZ 3 in this script does the work of encoding a letter:

```
shiftAZ 3 c = ltrFromInt((intFromLtr c + 3) `mod` 26)
```

The function first translates the character supplied as its argument to an integer between zero and twenty-five (intFromLtr c), then it adds three, computes the remainder in a division by twenty-six (to loop around to the beginning if the letter happened to be near the end of the alphabet), and finally converts the shifted number back to a letter (ltrFromInt(*all that stuff*)).

The functions that do the conversions between letters and integers use some intrinsic functions, toEnum and fromEnum, that do a slightly different conversion between letters and integers. The function toEnum will translate any argument of type Char into a value of type Int between zero and 255 (inclusive).[2] For any character *c* in the standard electronic alphabet, the Haskell formula fromEnum(*c*) denotes its ASCII code, which is a number between zero and 127. The function toEnum converts back to type Char. That is, for any ASCII character, toEnum(fromEnum(*c*))=*c*.

> ***ASCII character set***
>
> A standard known as ISO8859-1 specifying representations of a collection of 128 characters has been established by the International Standards Organization. These are usually called the ASCII characters—the American Standard Code for Information Interchange. ASCII, an older standard essentially consistent with ISO8859-1 but less inclusive of non-English alphabets, represents 128 characters (94 printable ones, plus the space-character, a delete-character, and 32 control-characters such as newline, tab, backspace, escape, bell, etc.) as integers between zero and 127

The designers of the ASCII character set arranged it so that the capital letters A to Z are represented by a contiguous set of integers, and the functions intFromLtr and ltrFromInt use this fact to their advantage: For any letter c,

fromEnum('A') ≤ fromEnum(*c*) ≤ fromEnum('Z')

1. Explained in the Haskell Report (see "monomorphism restriction").
2. Haskell uses type Int instead of Integer for these functions because Int is adequate for the range 0 to 255.

Therefore,

$$\text{fromEnum('A') - fromEnum('A')} \leq \text{fromEnum}(c) \text{ - fromEnum('A')} \leq \text{fromEnum('Z') - fromEnum('A')}$$

And, since the codes are contiguous, fromEnum('Z') - fromEnum('A') must be 25, which means

$$0 \leq \text{fromEnum}(c) \text{ - fromEnum('A')} \leq 25$$

Because of these relationships, you can see that intFromLtr will always deliver an integer between zero and 25 when supplied with a capital letter as its argument, and ltrFromInt just inverts this process to get back to the capital letter that the integer code came from.

> **fromEnum *and* toEnum**
>
> The class Enum includes Char, Bool, Int, and several other types. The function fromEnum converts from any of these types to Int, and toEnum goes in the other direction. Since there are several target types for toEnum to choose from, **explicit type declarations are often needed.**

The deciphering process is basically the same as the process of creating a ciphertext, except that instead of shifting by three letters forward (shiftAZ 3), you shift by three letters back in the alphabet (shiftAZ (-3)). So, the formula for the decipherJulius function is similar to the one for cipherJulius:

```
HASKELL DEFINITION •   cipherJulius  =  map (shiftAZ 3)
HASKELL DEFINITION •   decipherJulius  =  map (shiftAZ (-3))
```

The script, as formulated, takes some chances. It assumes that the supplied argument will be a sequence of capital letters — no lower case, no digits, etc. If someone tries to make a ciphertext from the plaintext "Veni vidi vici," it will not decipher properly:

```
HASKELL COMMAND •    cipherJulius "Veni vidi vici."
HASKELL RESPONSE •   YNWRWERMRWERLRK
HASKELL COMMAND •    decipherJulius "YNWRWERMRWERLRK"
HASKELL RESPONSE •   VKTOTBOJOTBOIOH
```

This is not good. This is not right. My feet stick out of … oh … sorry … lapsed into some old Dr Seuss rhymes … let me start again …

This is not good. It's ok for a program to have some restrictions on the kinds of data it can handle, but it's not ok for it to pretend that it's delivering correct results when, in fact, its delivering nonsense — especially if what you're expecting from the program is a ciphertext, which is supposed to look like nonsense, so you can't tell when the program is outside its domain.

One way to fix the program is to check for valid letters (that is, capital letters) when making the conversions between letters and integers. To do this, you need some way to provide alternatives in definitions, so that the intFromLtr function can apply the conversion formula when its argument is a capital letter and can signal an error[1] if its argument is something else.

Definitions present alternative results by prefacing each alternative with a guard. The guard is a formula denoting a Boolean value. If the value is True, the result it guards is delivered as the value

1. Any function can signal an error by delivering as its value the result of applying the function error to a string. The effect of delivering this value will be for the Haskell system to stop running the program and send the string as a message to the screen.

of the function. If not, the Haskell system proceeds to the next guard. The first guard to deliver True selects its associated formula as the value of the function. The last guard is always the keyword otherwise: if the Haskell system gets that far, it selects the alternative guarded by otherwise as the value of the function. One way to look at this is that each formula that provides an alternative value for the function is guarded by a Boolean value: they are a collection of guarded formulas.

A guard appears in a definition as a Boolean formula following a vertical bar ( |, like the one used in list comprehensions). After a guard comes an equal sign (=), and then the formula that the guard, if True, is supposed to select as the value of the function. Here's a function that delivers 1 if its argument exceeds zero and -1 otherwise:

```
HASKELL DEFINITION •   f x
HASKELL DEFINITION •     | x > 0     = 1
HASKELL DEFINITION •     | otherwise  =  -1
```

Try to apply this idea in the following script defining a safer version of the Caesar cipher system. In case the conversion functions intFromLtr and ltrFromInt encounter anything other than capital letters, use the error function to deliver their values. To test for capital letters, you can use the intrinsic function isUpper Char -> Bool, which delivers the value True if its argument is a capital letter, and False otherwise.

```
¿ HASKELL DEFINITION ?   import Char(isUpper)
¿ HASKELL DEFINITION ?   cipherJulius :: String -> String
¿ HASKELL DEFINITION ?   cipherJulius = map(shiftAZ 3)
¿ HASKELL DEFINITION ?   shiftAZ :: Int -> Char -> Char
¿ HASKELL DEFINITION ?   shiftAZ n c = ltrFromInt((intFromLtr c + n) `mod` 26)
¿ HASKELL DEFINITION ?   decipherJulius :: String -> String
¿ HASKELL DEFINITION ?   decipherJulius = map(shiftAZ (-3))
¿ HASKELL DEFINITION ?   intFromLtr :: Char -> Int
¿ HASKELL DEFINITION ?   intFromLtr c          -- you fill in the two-alternative definition
¿ HASKELL DEFINITION ?     | isUpper c  = fromEnum c - fromEnum 'A'
¿ HASKELL DEFINITION ?     | otherwise    = error "intFromLtr(non-letter) not allowed"
¿ HASKELL DEFINITION ?   ltrFromInt :: Int -> Char
¿ HASKELL DEFINITION ?   ltrFromInt n          -- again, you fill in the definition
¿ HASKELL DEFINITION ?     | isUpper c    = c
¿ HASKELL DEFINITION ?     | otherwise    = error "ltrFromInt(non-letter Int) not allowed"
¿ HASKELL DEFINITION ?     where
¿ HASKELL DEFINITION ?     c = toEnum(n + ord 'A')
```

2

```
HASKELL COMMAND •    cipherJulius "VENIVIDIVICI"
HASKELL RESPONSE •   "YHQLYLGLYLFL"
```

## ANSWERS — WORK OUT P.81Q BEFORE READING THIS PAGE

```
HASKELL COMMAND •   cipherJulius "Veni vidi vici."
HASKELL RESPONSE •  "Y
HASKELL RESPONSE •  Program error: intFromLtr(non-letter) not allowed
```

Another problem with the Caesar cipher system is that it uses a modern alphabet. The Roman alphabet in Caesar’s time had twenty-three letters, not twenty-six. The letters J, U, and W from the modern alphabet were not in the ancient one.[1] This, too, can be fixed by putting some additional alternatives in the conversion functions to squeeze out the gaps in that the omission of J, U, and W leave in the integer codes.

The idea is to check to see what range the letter is in A-I, K-T, V, or X-Z, then adjust by zero, one, two, or three, respectively. (Of course the clock arithmetic has to be done mod 23 rather than mod 26, too.) Checking for a range like A-I takes two tests: *c* >= ’A’ and *c* < ’J’. For compound formulas like this, Haskell provides the and-operator (&&). It takes two Boolean operands and delivers the value True if both operands are True and False otherwise. (Haskell also provides an or-operator (||) and a not-function (not), but they won’t be needed in this case.)

Try to work out the gaps in the following script, which encodes using the ancient Roman alphabet.

```
¿ HASKELL DEFINITION ?  cipherJulius :: String -> String
¿ HASKELL DEFINITION ?  cipherJulius = map(shiftRomanLetter 3)
¿ HASKELL DEFINITION ?  shiftRomanLetter :: Int -> Char -> Char
¿ HASKELL DEFINITION ?  shiftRomanLetter n c = romFromInt((intFromRom c + n) `mod` 23)
¿ HASKELL DEFINITION ?  intFromRom :: Char -> Int
¿ HASKELL DEFINITION ?  intFromRom c      -- you fill in the definition (5 alternatives)
¿ HASKELL DEFINITION ?     | c >='A' && c<'J'   = cInt
¿ HASKELL DEFINITION ?     | c >'J' && c<'U'    = cInt - 1
¿ HASKELL DEFINITION ?     | c >'U' && c<'W'    = cInt - 2
¿ HASKELL DEFINITION ?     | c >'W' && c<='Z'   = cInt - 3
¿ HASKELL DEFINITION ?     | otherwise    = error "non-Roman letter not allowed"
¿ HASKELL DEFINITION ?     where
¿ HASKELL DEFINITION ?     cInt = fromEnum c - fromEnum 'A'
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?  romFromInt :: Int -> Char
¿ HASKELL DEFINITION ?  romFromInt n      -- again, you fill in the definition
¿ HASKELL DEFINITION ?     | n>=intFromRom 'A' && n<=intFromRom 'I'  =fromASCII n
¿ HASKELL DEFINITION ?     | n>=intFromRom 'K' && n<=intFromRom 'T'  =fromASCII(n+1)
¿ HASKELL DEFINITION ?     | n>=intFromRom 'V' && n<=intFromRom 'V'  =fromASCII(n+2)
¿ HASKELL DEFINITION ?     | n>=intFromRom 'X' && n<=intFromRom 'Z'  =fromASCII(n+3)
```

1. A problem the cipher system doesn’t have that it might seem to have is that it can’t deal with spaces and punctuation. As it happens, the Romans didn’t use spaces and punctuation in their writing. I don’t know if they used lower case letters or not, but the all-upper-case messages look really Roman to me.

```
¿ HASKELL DEFINITION ?       | otherwise    = error "non-Roman Int not allowed"
¿ HASKELL DEFINITION ?       where
¿ HASKELL DEFINITION ?       fromASCII code = toEnum(code + fromEnum 'A')
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    decipherJulius :: String -> String
¿ HASKELL DEFINITION ?    decipherJulius = map(shiftRomanLetter (-3))                3
    HASKELL COMMAND •    cipherJulius "VENIVIDIVICI"
   HASKELL RESPONSE •    "ZHQMZMGMZMFM"
    HASKELL COMMAND •    decipherJulius "ZHQMZMGMZMFM"
   HASKELL RESPONSE •    "VENIVIDIVICI"
```

The Caesar cipher is not a very good one. You can look at even a short ciphertext, such as "ZHQMZMGMZMFM" and guess that M probably stands for a vowel. Ciphers like Caesar's are easy to break.[1]

## *Review Questions*

1 Guards in function definitions
- a hide the internal details of the function from other software components
- b remove some of the elements from the sequence
- c select the formula that delivers the value of the function ← *correct answer*
- d protect the function from damage by cosmic rays

2 The formula map reverse ["able", "was", "I"] delivers the value
- a ["I", "saw", "elba"]
- b ["elba", "saw", "I"] ← *correct answer*
- c ["I", "was", "able"]
- d ["able", "was", "I", "not"]

3 The formula map f xs delivers the value
- a f x
- b [f x | x <- xs] ← *correct answer*
- c f xs
- d [f xs]

4 Which of the following formulas is equivalent to the formula [g x y | y <- ys] ?
- a (map . g x) ys
- b (map g x) ys
- c map(g x y) ys
- d map(g x) ys ← *correct answer*

1. The article "Contemporary Cryptology: An Introduction," by James L. Massey, which appears in *Contemporary Cryptology, The Science of Information Integrity*, edited by Gustavus J. Simmons (IEEE Press, 1992), discusses methods of constructing good ciphers.

5 The following function delivers

```
HASKELL DEFINITION •    h  xs
HASKELL DEFINITION •         | xs == reverse xs      = "yes"
HASKELL DEFINITION •         | otherwise             = "no"
```

a "yes", unless xs is reversed
b "yes" if its argument is a palindrome, "no" if it's not ← *correct answer*
c "no" if xs is not reversed
d "yes" if its argument is written backwards, "no" if it's not

6 The following function

```
HASKELL DEFINITION •    s  x
HASKELL DEFINITION •         | x < 0    =  -1
HASKELL DEFINITION •         | x == 0   =   0
HASKELL DEFINITION •         | x > 0    =   1
```

a the value of its argument
b the negative of its argument
c a code indicating whether its argument is a number or not
d a code indicating whether its argument is positive, negative, or zero ← *correct answer*

7 Assuming the following definitions, which of the following functions puts in sequence of x's in place of all occurrences of a given word in a given sequence of words?

length :: [a] -> Int
length[$x_1$, $x_2$, …, $x_n$] = $n$

```
HASKELL DEFINITION •    rep  n  x  =  [ x | k <- [1 . . n]]
HASKELL DEFINITION •    replaceWord badWord word
HASKELL DEFINITION •      | badWord == word    = rep (length badWord) 'x'
HASKELL DEFINITION •      | otherwise   = word
```

a censor badWord = map (replaceWord badWord) ← *correct answer*
b censor badWord = map . replaceWord badWord
c censor badWord = replaceWord badWord . map
d censor badWord = map badWord . replaceWord

# *Modules as Libraries* 17

Encryption is an important application of computational power. It is also an interesting problem in information representation, and in that way is related to the question of representing numbers by numerals, which you have already studied. In fact, the numeral/number conversion software you studied earlier can be used to implement some reasonably sophisticated ciphers. So, constructing encryption software provides an opportunity to reuse some previously developed software.

Reusing existing software in new applications reduces the development effort required. For this reason, software reuse is an important idea in software engineering. Programming languages provide a collection of intrinsic functions and operations. Whenever you use one of these, you are reusing existing software. Similarly, when you package functions in a module, then import them for use in an application, you are reusing software. Modules and intrinsic functions provide repositories or **libraries** of software intended to be used in other applications.

This chapter presents some software for encryption, that is for encoding messages so that they will be difficult for people other than the intended receivers to decode. Encoding methods for this purpose are known as ciphers.

Substitution ciphers, in which there is a fixed replacement for each letter of the alphabet, are easy to break because the distribution of letters that occur in ordinary English discourse (or any other language) are known. For example, the letter E occurs most frequently in English sentences, followed by the letter T, etc. If you have a few sentences of ciphertext, you can compute the distribution of occurrence of each letter. Then you can guess that the most frequently occurring letter is the letter E, or maybe T, or one of the top few of the most frequently occurring letters. After guessing a few of the letter-substitutions by this method, you can break the code easily.[1]

The statistics on pairs of letters are also known. So, even if the cipher is designed to substitute a fixed new pair of letters for each pair that occur in the original message (maybe XQ for ST, RY for PO, and so on for all possible two-letter combinations), the cipher will not be hard to break. The code breaker will need access to a longer ciphertext, however, because the statistical differences among occurrences of different letter combinations are more subtle than for individual letters.

The same goes for substitution ciphers that use three-letter combinations, and so on. But, the longer the blocks of letters for which the cipher has a fixed replacement, the harder it is to break the code. Ciphers of this kind (that is, multi-letter substitution ciphers) make up a class known as block ciphers. The Data Encryption Standard (DES), which was designed and standardized in the 1970s, is a substitution cipher based on blocks of eight to ten letters, depending on how the message is represented. The method of computing the replacement combination, given the block of characters for which a new block is to be substituted, has sixteen stages of successive changes. It scrambles the message very successfully, but in principle, it is a multi-letter, substitution cipher.

To encode a message with the DES cipher, the correspondents agree on a key. The DES cipher then uses this key to compute the substitutions it will make to encrypt and decrypt messages. As

1. Edgar Allan Poe's story, *The Gold Bug*, contains an account of the breaking of a substitution cipher.

long as the key is kept secret, people other than the correspondents will have a very tough time decoding encrypted messages.

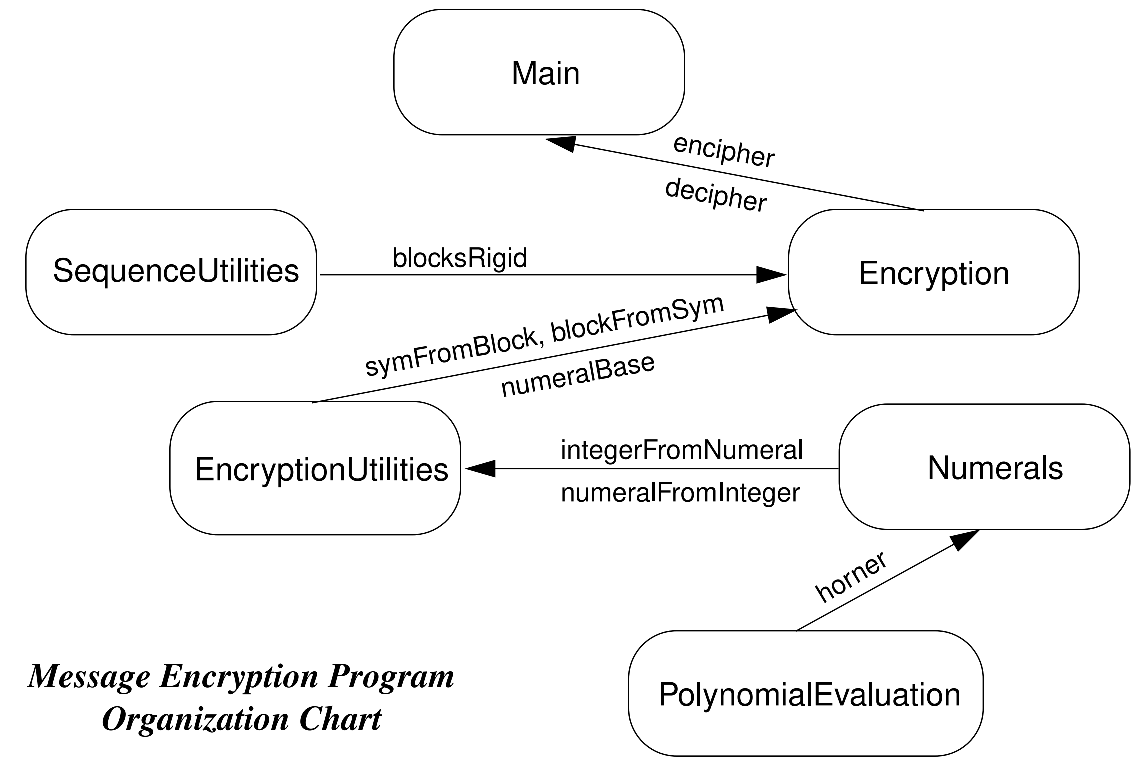
A block cipher is like the Caesar cipher, but on a larger alphabet. For example, if the message-alphabet consisted of capital letters and blanks, 27 symbols in all, and the block cipher substituted new three-letter combinations for the three-letter combinations in the message, then this would be a substitution cipher on an alphabet with 27×27×27 letters — that's 19,683 letters in all.

The following module, Encryption, contains software that implements a block cipher of this kind. It is not limited to three-letter combinations. Instead, it is parameterized with respect to the number of letters in the substitution-blocks. They can be of any length.

***DES Efficiency***

The encryption software developed in this lesson scrambles messages successfully, but requires much more computation than the DES procedure, which is carefully engineered for both security and efficient use of computational resources.

The overall structure of the program to be constructed is illustrated in the accompanying program organization chart. The Encryption module will import a function from the SequenceUtilities module (in the Appendix) to package messages into blocks. Each block will then be encrypted, with the help of some entities imported from an EncryptionUtilities module, which, itself, gets some help from the Numerals module developed previously. The Numerals module imported a function from the PolynomialEvaluation module. The program organization chart displays all these relationships. You can use it to help you keep track of what is going on as you work your way through this lesson.




***Message Encryption Program Organization Chart***

The encipher and decipher functions in the Encryption module are also parameterized with respect to the key. The correspondents can agree on any sequence of characters they like as a key for the software to use to encipher and decipher messages.

The cipher works like this. Given a fixed block size, it partitions the message into blocks of that length (say, for example, 20 characters per block). If the number of characters in the message is not an exact multiple of the block size, then the last block is padded with blanks to make it come out even.

Each block is then converted to a numeral (see page 74) by translating its block of characters into a block of integers. To do this, each letter in the alphabet that the message is written in is associated with an integer code (first letter coded as zero, second letter as on, etc.). The resulting numeral then denotes a number in standard, positional notation with a radix equal to the number of letters in the alphabet. The function integerFromNumeral (from the Numerals module), the numeral is converted into an integer, and it is this integer that is viewed as a character in the cipher alphabet.

The number of characters in the cipher alphabet varies with the chosen block size:

cipher-alphabet size = $\alpha^{\beta}$,
where $\alpha$ = message-alphabet size
$\beta$ = block size

The message alphabet consists of the printable characters of the ASCII character set (see "ASCII character set" page 78) plus the space, tab, and newline characters, for a total of 97 characters ($\alpha = 97$). If the correspondents were to choose a block size of one ($\beta = 1$) then the cipher alphabet would contain the same number of symbols as the message alphabet (97), which would produced a simple substitution cipher similar to the Caesar cipher. But, with a block size of five ($\beta = 5$), the number of symbols in the cipher alphabet goes up to several billion ($97^5 = 8{,}587{,}340{,}257$), and with a block size of twenty ($\beta = 20$) up to huge number with forty digits in its decimal numeral (enter the Haskell command (97::Integer)^20 if you want to see the exact number).

***character-blocks as numerals***

The encryption software essentially interprets each block of characters as a base-97 numeral. The "digits" in the numeral are ASCII characters.

Example, block-length 3, numeral "AbX":

AbX

denotes the cipher-alphabet symbol

$\text{code(A)}\times 97^2 + \text{code(b)}\times 97^1 + \text{code(X)}\times 97^0$

where code(A), code(b), and code(X) are numbers between zero and 96 computed from the ASCII codes for those characters.

After converting a block of characters in the original message to an integer (denoting a symbol in the cipher alphabet), an integer version of the key is added. (The integer version of the key is gotten by interpreting the key as a base-97 numeral, just as with blocks of characters from a message.) This sum denotes another symbol in the cipher alphabet, shifted from the original symbol by the amount denoted by the key (just as with the Caesar cipher, but on a larger scale: the remainder is computed modulo the number of characters in the cipher alphabet — that is $97^{\beta}$, where $\beta$ is the block size). And, finally, the shifted integer is converted back to a block of characters by reversing the process used to convert the block of characters in the original message to an integer.

```
HASKELL DEFINITION •  module Encryption
HASKELL DEFINITION •    (encipher, decipher)
HASKELL DEFINITION •    where
HASKELL DEFINITION •
HASKELL DEFINITION •    import EncryptionUtilities
HASKELL DEFINITION •    import SequenceUtilities
HASKELL DEFINITION •
HASKELL DEFINITION •    encipher, decipher :: Int -> String -> String  -> String
HASKELL DEFINITION •    encipher blockSize key =
HASKELL DEFINITION •      cipher blockSize (symFromBlock key)
HASKELL DEFINITION •    decipher blockSize key =
HASKELL DEFINITION •      cipher blockSize (- symFromBlock key)
HASKELL DEFINITION •
HASKELL DEFINITION •    cipher :: Int -> Integer -> String -> String
HASKELL DEFINITION •    cipher blockSize keyAsInteger =
HASKELL DEFINITION •      concat .                                              -- de-block
HASKELL DEFINITION •      map blockFromSym .            -- back to blocks (from symbols)
HASKELL DEFINITION •      map shiftSym .                                 -- encipher symbols
HASKELL DEFINITION •      map symFromBlock .      -- convert to cipher-alphabet symbol
HASKELL DEFINITION •      blocksRigid blockSize ' '                            -- form blocks
HASKELL DEFINITION •      where
HASKELL DEFINITION •      shiftSym n = (n + keyAsInteger) `mod` alphabetSize
HASKELL DEFINITION •      alphabetSize = numeralBase^blockSize                      1
```

In this way, encoding a message is a five-step process:

1 group characters in original message into blocks
2 convert each block to a symbol in the cipher alphabet
3 shift the cipher-alphabet symbol by the amount denoted by the key
4 convert each (shifted) cipher-alphabet symbol into a block of characters
5 string the blocks together into one string, which is the encoded message (ciphertext)

The function cipher in the Encryption module is defined as five-step composition of functions, one function for each step in the encoding process. The functions encipher and decipher both use the function cipher, one with a forward version of the key and the other with a backward version (just as in the Caesar cipher, where the forward key advanced three letters in the alphabet to find the substitute letter, and the backward key shifted three letter back in the alphabet). The structure of these functions matches their counterparts in the Caesar cipher software, except for the addition of the blocking and de-blocking concepts.

The Encryption module imports functions from a module called EncryptionUtilities to convert between blocks of ASCII characters (type String) and cipher-alphabet symbols (type Integer). It maps the block-to-symbol function (called symFromBlock) onto the sequence of blocks of the original message, then maps the symbol-shifter function (shiftSym) onto the sequence of symbols, and then maps the symbol-to-block function (blockFromSym) onto the sequence of shifted symbols to get back to blocks again.

> *append operator* **(++)**
> glues two sequences together
>
> "Thelma" ++ "Louise" *means* "ThelmaLouise"
>
> [1, 2, 3, 4] ++ [5, 6, 7] *means* [1, 2, 3, 4, 5, 6, 7]

The Encryption module also uses the variable numeralBase from the EncryptionUtilities module, which provides the size of the cipher alphabet (alphabetSize). The Encryption module needs this value to do the shifting modulo the size of the alphabet, so that symbols that would shift off the end of the cipher-alphabet are recirculated to the front.

The Encryption module also uses a function called blocksRigid from the module SequenceUtilities to build blocks of characters from the original message string. It uses an intrisic function, concat, to paste the blocks of the encoded message back into a single string.

The SequenceUtilities module appears in the Appendix. It is a library of several functions useful for building or converting sequences in various ways. The blocksRigid function takes three arguments: a block size, a pad, and a sequence to group into blocks. It groups the sequence into as many blocks as it takes to contain all of its elements. The last block will be padded at the end, if necessary, to make it the same size as the others (the second argument says what pad-character to use). For now, it's best to accept that this function works as advertised, but when you have some free time, you can take a look at the Appendix and try to understand it. The definition uses some intrinsic functions that you haven't studied. You can look them up in the *Haskell Report*.

> **concat :: [[a]] -> [a]**
>
> concat [[1,2,3], [4,5], [6,7,8,9]] = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>
> concat = foldr (++) [ ]

The intrinsic function, concat, which converts the blocks back into one long string works as if you had put an append operator (++) between each pair of blocks in the sequence. In fact, it could be defined in exactly that manner using a fold operator.

The Encryption module defines two functions for export: encipher and decipher. It also defines a private function, cipher, which describes the bulk of the computation (that's where the five-link chain of function compositions is). It imports two functions (symFromBlock and blockFromSym) and a variable (numeralBase) from the EncryptionUtilities module. These entities are not exported from the Encryption module, so a script importing the Encryption module would not have access to symFromBlock, blockFromSym, or numeralBase. This is by design: presumably a script importing the Encryption module would do so to be able to encipher and decipher messages; it would not import the Encryption module to get access to the utility functions needed to encipher and decipher messages. The additional functions would just clutter up the name space.

Now, take a look at the EncryptionUtilities module (see page 90). It defines four functions: symFromBlock, blockFromSym, integerCodeFromChar, and charFromIntegerCode.

The purpose of the functions integerCodeFromChar, and charFromIntegerCode is to convert between values of type Integer and blocks with elements of type Char. These functions make this conversion for individual elements, and then they are mapped onto blocks to make the desired conversion. The functions are defined in a manner similar to intFromLtr and ltrFromInt in cipherJulius (see page 80), except that the new functions are simpler because there is only one gap in the ASCII codes for the characters involved (the ancient Roman character set had three gaps).

The ASCII codes for the space character and the 94 printable characters are contiguous, running from 32 for space (fromEnum(' ')=32) up to 126 for tilde (fromEnum('~')=126). The only gap is between those characters and the other two in the character set the software uses for encoding messages, namely tab (ASCII code 9 — fromEnum('/t')=9) and newline (ASCII code 10 — fromEnum('/n')=10). Given this information, try to write the definitions of these functions that convert between integers and code-characters.

Try to write the other two functions, too. Their definitions can be constructed as a composition of one of the integer/numeral conversion-functions in the Numerals module (see page 74) and a mapped version of one of the integer/code-character conversion-functions. It might take you a while to puzzle out these definitions — the three-minute rule is a bit short here. But, if you can work these out, or even get close, you should feel like you're really getting the hang of this.

```
¿ HASKELL DEFINITION ?  module EncryptionUtilities
¿ HASKELL DEFINITION ?    (symFromBlock, blockFromSym, numeralBase)
¿ HASKELL DEFINITION ?    where
¿ HASKELL DEFINITION ?    import Numerals
¿ HASKELL DEFINITION ?    symFromBlock :: String -> Integer        -- you write the function
¿ HASKELL DEFINITION ?    symFromBlock =
¿ HASKELL DEFINITION ?      integerFromNumeral numeralBase .
¿ HASKELL DEFINITION ?      map integerCodeFromChar
¿ HASKELL DEFINITION ?    blockFromSym :: Integer -> String              -- again, you write it
¿ HASKELL DEFINITION ?    blockFromSym =
¿ HASKELL DEFINITION ?      map charFromIntegerCode .
¿ HASKELL DEFINITION ?      numeralFromInteger numeralBase
¿ HASKELL DEFINITION ?    integerCodeFromChar :: Char -> Integer     -- write this one, too
¿ HASKELL DEFINITION ?    integerCodeFromChar c
¿ HASKELL DEFINITION ?      | fromEnum c >= minCode =
¿ HASKELL DEFINITION ?                                fromIntegral(fromEnum c - minCode+2)
¿ HASKELL DEFINITION ?      | otherwise  = fromIntegral(fromEnum c - fromEnum '\t')
¿ HASKELL DEFINITION ?    charFromIntegerCode :: Integer -> Char             -- and this one
¿ HASKELL DEFINITION ?    charFromIntegerCode integerCode
¿ HASKELL DEFINITION ?      | intCode >=
¿ HASKELL DEFINITION ?                  numExtraCodes = toEnum(intCode + minCode - 2)
¿ HASKELL DEFINITION ?      | otherwise  = toEnum(intCode + fromEnum '\t')
¿ HASKELL DEFINITION ?      where
¿ HASKELL DEFINITION ?      intCode = fromIntegral integerCode
¿ HASKELL DEFINITION ?    maxCode, minCode, numExtraCodes :: Int
¿ HASKELL DEFINITION ?    maxCode = 126                           -- set of code-characters =
¿ HASKELL DEFINITION ?    minCode = 32          -- {tab, newline, toEnum 32 ... toEnum 126}
¿ HASKELL DEFINITION ?    numExtraCodes = 2
```

```
¿ HASKELL DEFINITION ?     numeralBase :: Integer
¿ HASKELL DEFINITION ?     numeralBase =
¿ HASKELL DEFINITION ?        fromIntegral(maxCode - minCode + 1 + numExtraCodes)     2
```

As you can see, the EncryptionUtilities module is rather fastidious about the differences between type Int and type Integer. The reason the issue arises is that the functions fromEnum and toEnum, which are used to convert between characters and ASCII codes, deal with type Int. They may as well, after all, because all the integers involved are between zero and 255, so there is no need to make use of the unbounded capacity of type Integer. Type Int is more than adequate with its range limit of $2^{29}$ - 1, positive or negative (see page 58).

However, Int is definitely *not adequate* for representing the integers that will occur in the cipher alphabet. These numbers run out of the range of Int as soon as the block size exceeds four.[1] So, the computations specified by the integer/numeral conversion functions of the Numerals module must be carried out using type Integer. For this reason, the functions integerCodeFromChar and charFromIntegerCode use type Integer on the integer side of the conversion and type Int on the character side. To do this it is necessary to convert between Int and Integer, and an intrinsic function is available to do this: fromIntegral. The function fromIntegral, given an argument in the class Integral (that is, an argument of type Int or Integer), delivers a number of the appropriate type for the context of the invocation.

The following script encrypts a maxim from Professor Dijkstra, which appeared in an open letter in 1975 and later in an anthology of his writings.[2] It imports the Encryption module and uses its exported functions. As the commands demonstrate, enciphering the message, then deciphering it gets back to the original (plus a few blanks at the end, depending on how the blocking goes).

```
HASKELL DEFINITION •   import Encryption
HASKELL DEFINITION •
HASKELL DEFINITION •   maximDijkstra, ciphertext, plaintext :: String
HASKELL DEFINITION •   maximDijkstra =
HASKELL DEFINITION •      "Besides a mathematical inclination, an exceptionally\n" ++
HASKELL DEFINITION •      "good mastery of one's native tongue is the most vital\n" ++
HASKELL DEFINITION •      "asset of a competent programmer.\n"
HASKELL DEFINITION •   ciphertext = encipher blockSize key maximDijkstra
HASKELL DEFINITION •   plaintext = decipher blockSize key ciphertext
HASKELL DEFINITION •
HASKELL DEFINITION •   key :: String
HASKELL DEFINITION •   key = "computing science"
HASKELL DEFINITION •   blockSize :: Int
HASKELL DEFINITION •   blockSize = 10                                              3
```

Using this program involves displaying messages that may be several lines long. If these messages are displayed directly as strings, they will be represented in the form that strings are denoted in Haskell programs. In particular, newline characters will appear in the form “\n”, and, of course the string will be enclosed in quotation marks.

---

1. $97^5 > 2^{29} - 1$
2. *Selected Writings on Computing: A Personal Perspective*, Edsger W. Dijkstra (Springer-Verlag, 1982).

The putStr directive makes it possible to display the contents of a string, rather than the Haskell notation for the string. This leaves off the surrounding quotation marks and interprets special characters in their intended display form. For example, the newline will be displayed by starting a new line and displaying subsequent characters from that point. The following commands, making use of the above program, use putStr to improve the display in this way.

> **putStr *directive***
> putStr :: String -> IO( )
>
> Causes the contents of the string specified as its argument to be displayed on the screen with each character interpreted in the normal way (e.g., newline characters start new lines, tabs cause spacing, etc.).

HASKELL COMMAND • putStr maximDijkstra — *display contents of string*
HASKELL RESPONSE • Besides a mathematical inclination, an exceptionally
HASKELL RESPONSE • good mastery of one's native tongue is the most vital
HASKELL RESPONSE • asset of a competent programmer.
HASKELL COMMAND • putStr ciphertext — *display contents of encrypted string*
HASKELL RESPONSE • 2Mv^IPYqEg]Iw]JXHdNIQT# … *and a bunch more gobbledygook …*
HASKELL COMMAND • putStr plaintext — *display contents of deciphered string*
HASKELL RESPONSE • Besides a mathematical inclination, an ... *etc. (as above) …*

## *Review Questions*

1 Software libraries
- a contain functions encapsulated in modules
- b provide a way to package reusable software
- c both of the above ← *correct answer*
- d none of the above

2 A module that supplies reusable software should
- a export all of the functions it defines
- b import all of the functions it defines
- c export reusable functions, but prevent outside access to functions of limited use ← *correct answer*
- d import reusable functions, but avoid exporting them

3 The formula concat ["The", "Gold", "Bug"] delivers
- a "The Gold Bug"
- b ["The", "Gold", "Bug"]
- c "TheGoldBug" ← *correct answer*
- d [["The], ["Gold"], ["Bug"]]

4 Encryption is a good example to study in a computer science course because
- a it is an important use of computers
- b it involves the concept of representing information in different ways
- c both of the above ← *correct answer*
- d well … really … it's a pretty dumb thing to study

5 The DES cipher is a block cipher. A block cipher is
- a a substitution cipher on a large alphabet ← *correct answer*
- b a rotation cipher with scrambled internal cycles
- c less secure than a substitution cipher
- d more secure than a substitution cipher

## ANSWERS — WORK OUT P.92Q BEFORE READING THIS PAGE

6 Professor Dijkstra thinks that in the software development profession
   a mathematical ability is the only really important asset that programmers need
   b the ability to express oneself in a natural language is a great asset to programmers ← *correct answer*
   c mathematical ability doesn't have much influence on a programmer's effectiveness
   d it's a waste of time to prove, mathematically, the correctness of program components

# Interactive Keyboard Input and Screen Output 18

Input and output are managed by the operating system. Haskell communicates with the operating system to get these things done. Through a collection of intrinsic functions that deliver values of IO type, Haskell scripts specify requests for services from the operating system. The Haskell system interprets IO type values and, as part of this interpretation process, asks the operating system to perform input and output.

For example, the following script uses the intrinsic function putStr to display the string "Hello World" on the screen:

```
HASKELL DEFINITION •   main = putStr "Hello World"
```

By convention, Haskell scripts that perform input and/or output define a variable named main in the main module. Entering the command main then causes Haskell system to compute the value of the variable main. That value, itself, is of no consequence. But, in computing the value, Haskell uses the operating system to perform the input/output specified in the script.

```
HASKELL COMMAND •   main
OP SYS RESPONSE •   Hello World
```

When the value delivered by a Haskell command is of IO type (e.g., main) the Haskell system does not respond by printing the value. Instead it responds by sending appropriate signals to the operating system. In this case, those signals cause the operating system to display the string "Hello World" on the screen. This is an output directive performed by the operating system.

Input directives are another possibility. For example, Haskell can associate strings entered from the keyboard with variables in a Haskell program.

Any useful program that reads input from the keyboard will also contain output directives. So, a script containing an input directive will contain one or more output directives, and these directives will need to occur in a certain sequence. In Haskell, such sequences of input/output directives are specified in a do-expression.

```
HASKELL DEFINITION •   main =
HASKELL DEFINITION •     do
HASKELL DEFINITION •       putStr "Please enter your name.\n"
HASKELL DEFINITION •       name <- getLine
HASKELL DEFINITION •       putStr("Thank you, " ++ name ++ ".\n" ++
HASKELL DEFINITION •         "Have a nice day (:-)\n"   )
```
6

```
HASKELL COMMAND •   main
OP SYS RESPONSE •   Please enter your name.
OP SYS ECHO •       Fielding Mellish
OP SYS RESPONSE •   Thank you, Fielding Mellish.
OP SYS RESPONSE •   Have a nice day (:-)
```

*Haskell-induced output*

*echo by operating system of keyboard entry*

## ANSWERS — WORK OUT P.94Q BEFORE READING THIS PAGE

A do-expression consists of the keyword do followed by a sequence of input/output directives. The example presented here contains a sequence of three such directives:

```
1    putStr "Please enter your name.\n"
2    name <- getLine
3    putStr("Thank you, " ++ name ++ ".\n" ++
            "Have a nice day (:-)\n"   )
```

*causes operating system to display a line on the screen*

*causes operating system to read a line entered at the keyboard — the string entered becomes the value of the variable* name

*causes operating system to display two lines on screen*

The first directive sends the string "Please enter your name.\n" to the screen. Since the string ends in a newline character, the string "Please enter your name." appears on the screen, and the cursor moves to the beginning of the next line. The second directive (name <- getLine) reads a line entered from the keyboard and associates the sequence of characters entered on the line[1] with the variable specified on the left side of the arrow (<-), which in this example is the variable called name. Any subsequent directive in the do-expression can refer to that variable, but the variable is not accessible outside the do-expression. And finally, the third directive sends a string constructed from name (the string retrieved from the keyboard) and some other strings ("Thank you, ", a string containing only the newline character, and "Have a nice day (:-)\n").

When the name is entered, the Haskell system builds a string from the characters entered and associates that string with the variable called name. While it is doing this, the operating system is sending the characters entered to the screen. This is known as echoing the input, and it is the normal mode of operation; without echoing, people could not see what they were typing.

> ***echo***
>
> Operating systems normally send characters to the screen as they are entered at the keyboard. This is known as echoing the characters, and it is usually the desired form of operation. Most operating systems have a way to turn off the echo when that is more desirable, such as for password entry. Haskell provides a directive to control echoing. See the Haskell Report.

When the string is complete, the person at the keyboard enters a newline character. This terminates the getLine directive (a newline is what it was looking for). And, since the operating system echoes the characters as they come in, the newline entry causes the cursor on the screen to move to the beginning of the line following the name-entry line.

The third directive sends a string to the screen containing two newline characters. In response to this signal, two new lines appear on the screen. You can see by looking at the script that the first one ends with a period, and the second one ends with a smiley-face.

What happens to the rest of the characters? The ones entered at the keyboard after the newline? Well, this particular script ignores them. But, if the sequence of input/output directives in the do-expression had contained other getLine directives, the script would have associated the strings entered on those lines with the variables specified in the getLine directives.

The sequence of input/output directives in the do-expression could, of course, include more steps. The following script retrieves two entries from the keyboard, then incorporates the entries into a screen display, and finally retrieves a sign-off line from the keyboard.

1. That is, all the characters entered up to, but not including, the newline character. The newline character is discarded

```
HASKELL DEFINITION •    main =
HASKELL DEFINITION •      do
HASKELL DEFINITION •        putStr "Please enter your name: "
HASKELL DEFINITION •        name <- getLine
HASKELL DEFINITION •        putStr "And your email address, please: "
HASKELL DEFINITION •        address <- getLine
HASKELL DEFINITION •        putStr(unlines[
HASKELL DEFINITION •          "Thank you, " ++ name ++ ".",
HASKELL DEFINITION •          "I'll send your email to " ++ address,
HASKELL DEFINITION •          "Press Enter to sign off."])
HASKELL DEFINITION •        signOff <- getLine
HASKELL DEFINITION •        return()                                        7
HASKELL COMMAND •       main                                   underline shows OS echo
HASKELL RESP / OS ECHO • Please enter your name: Captain Ahab
HASKELL RESP / OS ECHO • And your email address, please: cap@mobydick.org
HASKELL RESPONSE •      Thank you, Captain Ahab.
HASKELL RESPONSE •      I'll send your email to cap@mobydick.org
HASKELL RESPONSE •      Press Enter to sign off.
OP SYS ECHO •                         <---- echo of Enter-key from keyboard
```

There are a few subtleties going on with newline characters. The string sent to the screen by the first directive does not end with a newline. For that reason, the cursor on the screen remains at the end of the string "Please enter your name: " while waiting for the name to be entered.

After completing the name entry, the person at the keyboard presses the Enter key (that is, the newline character). The operating system echoes the newline to the screen, which moves the cursor to the beginning of the next line, and the Haskell system completes its performance of the getLine directive. Then, a similar sequence occurs again with the request for an email address.

Next, the putStr directive sends a three-line display to the screen. This string is constructed with an intrinsic function called unlines. The unlines function takes a sequence of strings as its argument and constructs a single string containing all of the strings in the argument sequence, but with a newline character inserted at the end of each them. In this case, there are three strings in the argument, so the result is a string containing three newline characters. This string, displayed on the screen, appears as three lines.

> **unlines :: [String] -> String**
> unlines = concat . map (++ "\n")
>
> unlines ["line1", "line2", "line3"] =
> "line1\nline2\nline3\n"]
>
> unlines *takes a sequence of strings and delivers a string that appends together the strings in the sequence, each followed by a newline character.*

The last input/output directive in the do-expression is another getLine. This one simply waits for the entry of a newline character. Because the variable that gets the value entered (signOff) is not used elsewhere in the script, all characters entered up to and including the expected newline are, effectively, discarded.

## *Review Questions*

1 Values of IO type
- a are in the equality class Eq
- b specify requests for operating system services ← *correct answer*
- c represent tuples in a unique way
- d describe Jovian satellites

2 Which of the following intrinsic functions in Haskell causes output to appear on the screen?
- a concat :: [[any]] -> [any]
- b putStr :: String -> IO () ← *correct answer*
- c printString :: Message -> Screen
- d getLine :: IO String

3 What will be the effect of the command main, given the following script?

```
HASKELL DEFINITION •    main =
HASKELL DEFINITION •        do    putStr "Good "
HASKELL DEFINITION •              putStr "Vibrations\n"
HASKELL DEFINITION •              putStr "   by the Beach Boys\n"
```

- a one line displayed on screen
- b two lines displayed on screen ← *correct answer*
- c three lines displayed on screen
- d audio effects through the speaker

4 What will be the effect of the command main, given the following script?

```
HASKELL DEFINITION •    main =
HASKELL DEFINITION •        do    putStr "Please enter your first and last name (e.g., John Doe): "
HASKELL DEFINITION •              firstLast <- getLine
HASKELL DEFINITION •              putStr (reverse firstLast)
```

- a display of name entered, but with the last name first
- b display of last name only, first name ignored
- c display of last name only, spelled backwards
- d display of name spelled backwards ← *correct answer*

5 How should the last input/output directive in the preceding question be changed to display the first name only?
- a putStr(take 1 firstLast)
- b putStr(drop 1 firstLast)
- c putStr(takeWhile (/= ' ') firstLast) ← *correct answer*
- d putStr(dropWhile (/= ' ') firstLast)

ANSWERS — WORK OUT P.97Q BEFORE READING THIS PAGE

# Interactive Programs with File Input/Output 19

Software can interact with people through the keyboard and the screen, and you have learned how to construct software that does this (see “Interactive Keyboard Input and Screen Output” on page 93). Since the screen is a highly volatile device, information displayed on it doesn’t last long — it is soon overwritten with other information. The computer system provides a facility known as the file system for recording information to be retained over a period of time and retrieved as needed. By interacting with the file system, software can retrieve information from files that were created at an earlier time, possibly by other pieces of software, and can create files containing information for processing at a later time.

Suppose, for example, you wanted to write a Haskell script that would record, in a file that could be accessed at a later time, a line of text entered at the keyboard. The script would begin by displaying a message on the screen asking the person at the keyboard to enter the line of text. Then it would write a file consisting of that line.

```
HASKELL DEFINITION •   main =
HASKELL DEFINITION •     do
HASKELL DEFINITION •       putStr(unlines["Enter one line."])
HASKELL DEFINITION •       lineFromKeyboard <- getLine
HASKELL DEFINITION •       writeFile filename lineFromKeyboard
HASKELL DEFINITION •       putStr("Entered line written to file \"" ++ filename ++ "\"")
HASKELL DEFINITION •     where
HASKELL DEFINITION •     filename = "oneLiner.txt"
```
1

**writeFile :: String -> String -> IO()**

writeFile filename contents

*name of file to be created*

*entire contents of file to be created*

Writing the file is accomplished through an output directive called writeFile. The first argument of writeFile is a string containing the name of the file to be created, and the second argument is the string to be written to the file. In this case, the string contains only one line, but it could contain any number of lines. For example, the following script writes a file containing three lines.

```
HASKELL DEFINITION •   main =
HASKELL DEFINITION •     writeFile "restaurant.dat" (unlines pepes)
HASKELL DEFINITION •     where
HASKELL DEFINITION •       pepes = ["Pepe Delgados", "752 Asp", "321-6232"]
```

**readFile :: String -> IO String**

contents <- readFile filename

*entire contents of file, retrieved as needed*

*name of file to be retrieved*

So, the writeFile directive creates a file of text. The readFile directive does the reverse: it retrieves the contents of an existing file. In a script, the readFile directive is used much as getLine is used, except that instead of retrieving a single line from the screen, readFile retrieves the entire contents of a file.

The contents are retrieved on an as-needed basis, following the usual Haskell strategy of lazy evaluation. But, the script accesses the file contents through the variable named on the left of the arrow (<-) preceding the readFile directive, and any input/output command following the readFile command in the do-expression containing it can refer to that variable.

To illustrate the use of file input/output, consider the problem of encrypting the text contained in a file. That is, suppose you want to retrieve a text from a file, encrypt it, then create a new file containing an encrypted version of the file contents.

The following script solves this problem by first asking for the name of a file from the keyboard, confirming it, then asking for a sequence of characters to use as an encryption key. When the key is entered, the script reads the contents of the file (that is, the plaintext), enciphers it using a function from the Encryption module developed earlier (page 87), and writes the encrypted message in a file with the same name as the one containing the plaintext, but with an extended name (".ctx", for ciphertext, is added to the filename).

```
HASKELL DEFINITION •   import Encryption(encipher)
HASKELL DEFINITION •
HASKELL DEFINITION •   main =
HASKELL DEFINITION •     do
HASKELL DEFINITION •       filename <- getFilename
HASKELL DEFINITION •       confirmFilename filename
HASKELL DEFINITION •       key <- getKey
HASKELL DEFINITION •       confirmKey
HASKELL DEFINITION •       putStr(msgReading filename)
HASKELL DEFINITION •       plaintext <- readFile filename
HASKELL DEFINITION •       putStr msgComputing
HASKELL DEFINITION •       writeFile (outFile filename) (encipher blockSize key plaintext)
HASKELL DEFINITION •       putStr (msgSignOff(outFile filename))
HASKELL DEFINITION •
HASKELL DEFINITION •   getFilename =
HASKELL DEFINITION •     do
HASKELL DEFINITION •       putStr msgEnterFilename
HASKELL DEFINITION •       filename <- getLine
HASKELL DEFINITION •       return filename
HASKELL DEFINITION •
HASKELL DEFINITION •   confirmFilename filename = putStr (msgThxForFilename filename)
HASKELL DEFINITION •
HASKELL DEFINITION •   getKey =
HASKELL DEFINITION •     do
HASKELL DEFINITION •       putStr msgEnterKey
HASKELL DEFINITION •       key <- getLine
HASKELL DEFINITION •       return key
HASKELL DEFINITION •
HASKELL DEFINITION •   confirmKey = putStr msgThxForKey
HASKELL DEFINITION •
HASKELL DEFINITION •   msgEnterFilename = "Enter name of file containing plaintext: "
HASKELL DEFINITION •
```

```
HASKELL DEFINITION •   msgThxForFilename filename =
HASKELL DEFINITION •     unlines[
HASKELL DEFINITION •       "Thank you",
HASKELL DEFINITION •       "   ... will read plaintext from " ++ filename,
HASKELL DEFINITION •       "   ... and write ciphertext to " ++ outFile filename]
HASKELL DEFINITION •
HASKELL DEFINITION •   msgEnterKey = "Enter key: "
HASKELL DEFINITION •
HASKELL DEFINITION •   msgThxForKey =
HASKELL DEFINITION •     unlines[
HASKELL DEFINITION •       "Thank you ...",
HASKELL DEFINITION •       "   ... will use key, then throw into bit-bucket"]
HASKELL DEFINITION •
HASKELL DEFINITION •   msgReading filename =
HASKELL DEFINITION •     unlines["Reading plaintext from " ++ filename]
HASKELL DEFINITION •
HASKELL DEFINITION •   msgComputing = unlines["  ... computing ciphertext"]
HASKELL DEFINITION •
HASKELL DEFINITION •   msgSignOff filename =
HASKELL DEFINITION •     unlines["  ... ciphertext written to " ++ filename]
HASKELL DEFINITION •
HASKELL DEFINITION •   outFile filename = filename ++ ".ctx"
HASKELL DEFINITION •
HASKELL DEFINITION •   blockSize :: Int
HASKELL DEFINITION •   blockSize = 10
```
2

Some of the functions in this script retrieve input from the keyboard (getLine or readFile) and need to deliver the input as their IO String values. This can be accomplished from a do-expression by using the return directive. When the return directive at the end of a do-expression containing input/output directives is supplied with an argument that is a String, then the do-expression delivers a value that can be used in the same way as a value delivered by getLine or readFile. In this way, you can write functions that do specialized sorts of input directives, such prompting for and retrieving the filename (getFilename) and the key (getKey) in the above script.

# ANSWERS — WORK OUT P.100Q BEFORE READING THIS PAGE

The following interactive session illustrates the use of the preceding script. Its effects, other than the interaction you seen on the screen, are the file reading and writing shown in the diagram.

| | |
|---|---|
| *HASKELL COMMAND* • | main |
| *HASKELL RESP / OS ECHO* • | Enter name of file containing plaintext: sussman.txt |
| *HASKELL RESPONSE* • | Thank you |
| *HASKELL RESPONSE* • | Reading plaintext from sussman.txt |
| *HASKELL RESPONSE* • | Writing ciphertext to sussman.txt.ctx |
| *HASKELL RESP / OS ECHO* • | Enter key (best if 10 or more characters): functional programming |
| *HASKELL RESPONSE* • | Thank you ... |
| *HASKELL RESPONSE* • | ... will use key, then throw into bit-bucket |

*underline indicates OS echo*

*file:* sussman.txt

The ultimate value generated by computer scientists is
the invention of languages for describing processes ...
What computer science delivers, has delivered, and is
continuing to and will be developing in the future, are
methods of describing complicated processes such that
what used to take hundreds of pages of English text
to describe will take a few lines in a formal language.
Formal is important, because it is possible for us to
understand it, and to communicate it quickly, and for
it not to be ambiguous and perhaps for us to run it.

Gerald Sussman (Comm ACM, Nov 1991)

*reads this file*

*writes this file*

3

*file:* sussman.txt.ctx

```
HYMuX[cT]IhUjjD[dOqOY_NfDcTNqJlrLcP_d_UZudL]
H]cTc\frRfccWOqQbgNbWX^XqWYrUUQVdLWMfrOc
TpSPcKfZK]QUp[bWWV\gHaoy~wtHQUVpRZ]XieNedb
RTUVWUjXH[XaUZf}j\DapOUT]gNfHR{lQVWrRfcR^
YdQbfRbIpcYqIbTjkL[ZlRMuUNjH[^[YVZrRadcWOqNi
e^fG{pLbMt^NhK^S]qWYrMYVRaTRQbWjWR\_WYK
UeNWd_aZSMgdNfdbdNWihYJgcfWLciidNWdc]ldI_Uj\
X]S]ULfrXYd_PRU[u`Nu(]VWY[[r]Y[bo_^iXV\WUXQ
Oq_]]TuWPZOqIuWNjd[XYU[uZVuCpUZbUU\j`D]V`Q
OY
i:Ra\L[i]cj]P_^]dIbduuETRLe[XrRgdXal`WgdRVOSpQ_Z
```

*... etc. ...*

*newline characters not shown accurately*

# *Fractional Numbers* 20

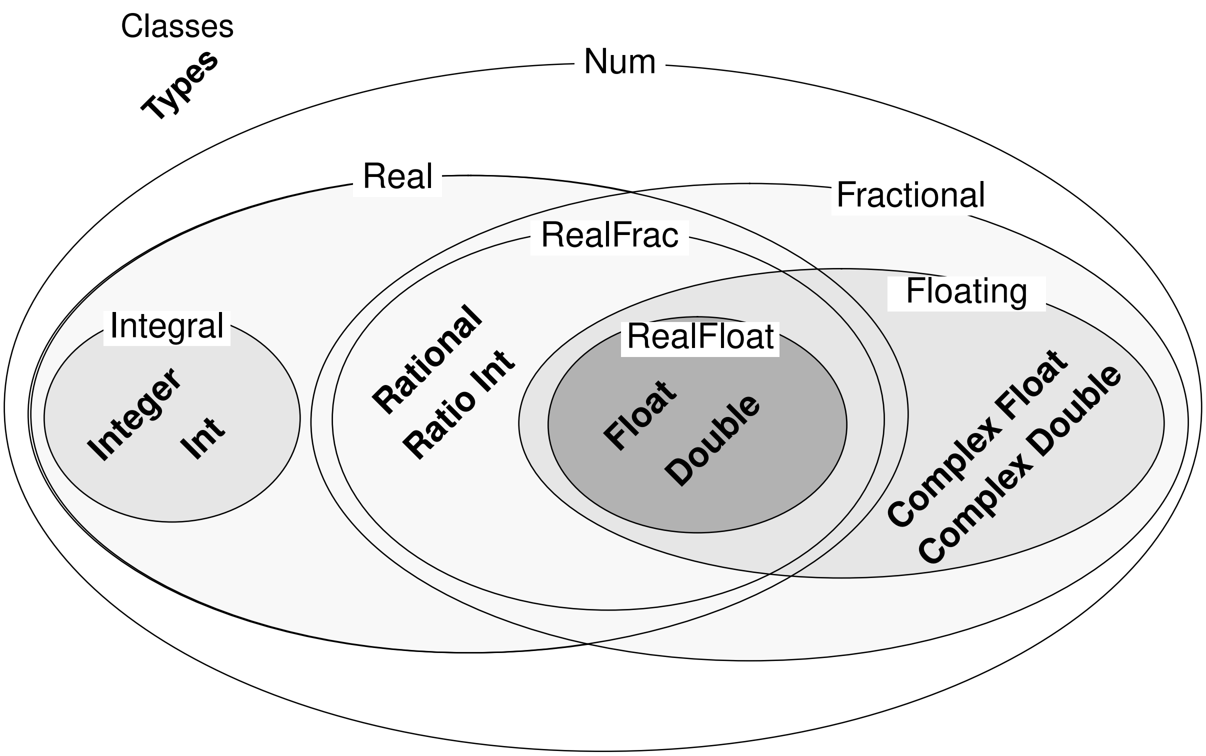
All of the software developed so far in this textbook has dealt primarily with strings or, in some cases integral numbers, but even then with some sort of string processing as an ultimate goal. This chapter discusses a computing application that makes use of non-integral numbers — that is, numbers in the Haskell class Fractional. This class encompasses the intrinsic types in Haskell that represent numbers with fractional parts.

There are six intrinsic types in this class. Two of them, the Complex types, are used to build models of many phenomena studied in mathematics, physics, and engineering. You can learn about Complex types on your own, using the Haskell Report as a reference, if you decide to build software that requires them. The types used in the examples in this chapter fall into the subclass RealFrac.

The term "real number," in mathematics, refers to the kinds of numbers used to count things and measure things. They can be whole numbers, which Haskell represents by the class Integral, or numbers with fractional parts, which Haskell represents by the class RealFrac. The most commonly used types in this class are Float and Double.

Numbers of type Float and Double have two parts: a **mantissa** and an **exponent**. The mantissa can be viewed as a whole number with a fixed number of digits (maybe decimal digits, but probably binary digits — the Haskell system uses a radix compatible with the computer system's instruction set), and the exponent as another whole number that specifies a scaling factor for the

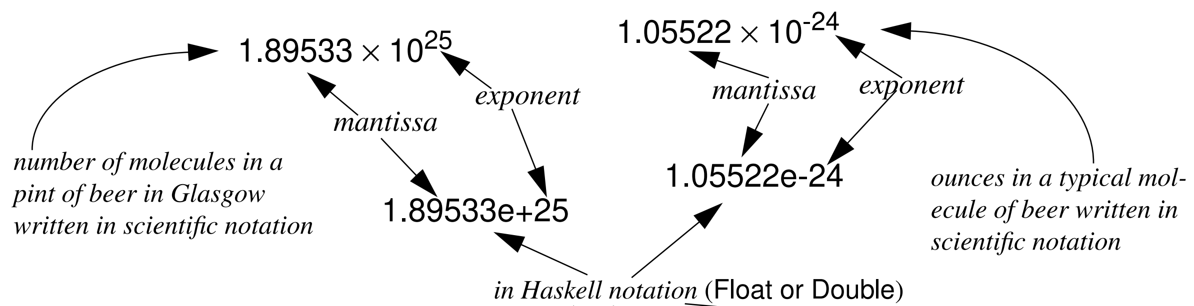
***The Class of Numbers***



mantissa. The scaling factor will be a power of the radix of the number system used to represent the mantissa. In effect, the exponent moves the decimal point in the mantissa (or binary point … or whatever) to the right or left. The decimal point moves to the right when the exponent is positive and to the left when it is negative. This is called a **floating point** representation. It is the basis of most numerical computations in scientific computing. All computers intended for use in studying models of scientific phenomena include, in their basic instruction sets, operators to do arithmetic with floating point numbers at speeds ranging from thousands of floating point operations per second on inexpensive systems to billions per second on computers intended for large-scale scientific computation.

***floating point numbers and scientific notation***

Because the numbers that occur in measuring physical phenomena range from very small to very large, and because the precision with which they can be measured runs from a few decimal digits to many, but usually not more than ten or twenty decimal digits of precision, measurements are often expressed as numbers that specify quantities in the form of a mantissa times a power of ten. In effect, the power of ten shifts the decimal point to scale the measured quantity  appropriately. This scheme for denoting numbers, known as scientific notation, is a form of floating point representation.




*bonafides: Avogadro's number* = 6.0221367e23*, molecular weight of $H_2O$* = 18.01528*, grams per ounce* = 28.24952

The difference between type Float and type Double is that numbers of type Double carry about twice the precision of numbers of type Float (that is, their mantissas contain twice as many digits). Both types are denoted in Haskell scripts by a decimal numeral specifying the mantissa and another decimal numeral specifying the exponent. The mantissa portion is separated from the exponent portion by the letter e.

The exponent portion is optional. It may be either negative (indicated by a minus sign beginning the exponent) or positive (indicated by a plus sign beginning the exponent or by the absence of a sign on the exponent). If the exponent part is present, then the mantissa must contain a decimal point, and that decimal point must be imbedded between two digits. Negative numbers have a minus sign at the beginning of the mantissa.

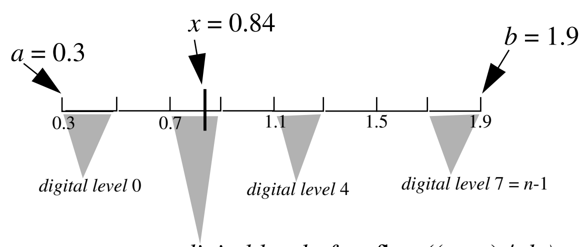
This chapter illustrates the use of fractional numbers through an example that builds a graphical representation of a numeric function. That is, given a function that, when supplied with a fractional number, delivers a new fractional number, the software will deliver a string that represents the curve the function describes. When printed, this string will look like a graph of the function.[1]

1. Not a very good picture of the graph, really. It will be printed as ordinary text, so the resolution (distance between discrete points on the display device) will be poor. But, in principle, the ideas developed in the chapter could be applied to a graphical display device capable of any level of resolution.

A key step in the computation of a graphical representation of a numeric function is the conversion of analog values to digital values. The plotting device is a printer or screen, which has a certain number of positions along the horizontal axis in which it can display marks, and, likewise, a discrete resolution in the vertical direction. A printer is, in this sense, a digital display device.

Analog display devices are not limited to certain fixed display points. In principle, an analog display device would be able to display a point anywhere within a given range.[1] The numeric function whose graph will be plotted has an analog character. It's input will be a fractional number, and its output will be a fractional number. Both numbers will be of high enough precision that it is reasonable to view them as analog measurements. The software will have to convert each analog measurement into a digital level that represents a position in which a printer can make a mark.

***analog to digital conversion***




| | |
|---|---|
| $n$ | —digital levels = {0, 1, ... $n$-1} |
| $x$ | — analog measurement |
| $a < x < b$ | — analog range |
| $dx$ | — stepsize, analog domain |

$n = 8$

$dx = \frac{b-a}{n} = 0.2$

*digital level of x*= floor(($x$ - $a$) / $dx$) = 2

Suppose the analog measurements $x$ fall in the range $a < x < b$, for some fractional numbers $a$ and $b$, and the available digital levels are {0, 1, 2, …, $n$-1} for some integral number $n$. The idea is to divide the analog range into $n$ segments, label the segments from smallest to largest, 0, 1, … $n$-1, and figure out which segment $x$ falls in. The label of that segment will be the digital level of the analog measurement $x$.

**floor ::**
**(RealFrac x, Integral n) => x -> n**
floor x = largest integer not exceeding x

There is an arithmetic formula that produces the digital level from the analog measurement $x$, given the analog range ($a$, $b$) and the number of digital levels $n$. It works like this: divide $x$ - $a$, which is the distance between x and the low end of the analog range, by $dx = (b - a) / n$, which is the length of the segments in the analog range corresponding to the digital levels ($dx$ is called the step size in the analog domain), then convert the quotient to a whole number by dropping down to the next smaller integer (if by chance the quotient falls on an integral boundary, just use that integer as the converted quotient — this next-lower-integer, or, more precisely, the largest integer not exceeding $x$, is known as the floor of $x$). The whole number delivered by this process is the digital level of the analog measurement $x$.

digital level of $x$ = floor(($x$ - $a$) / $dx$)

This formula always works properly when the computations are exact. Floating point numbers, however, involve approximate arithmetic because the precision of the mantissa is limited. Impre-

1. In practice, this will not be so. Any physical device is capable of a certain amount of precision. The real difference between digital devices and analogue devices is that digital representations are exactly reproducible. You can make an exact copy of a digital picture. Analog representations, on the other hand, are only approximately reproducible. A copy will be almost the same, but not exactly.

cise arithmetic causes no problems for most of the range of values of the analog measurement $x$. At worst, the digital level is off by one when $x$ is very close to a segment boundary — no big deal. No big deal, that is, unless off by one can put the digital level outside the set of $n$ possible digital levels {0, 1, 2, …, $n$-1}.

If that happens, it's a disaster, because the software will need to use the digital level to control a digital device that cannot operate with digital levels outside its expectations. So, it is best to make a special case in the calculation when $x$ is near the low end of the range, $a$, or near the high end of the range, $b$.

These ideas are put together in the following definition of the function digitize. It selects a special formula to avoid the out-of-range disaster when the analog value is within a half-step of either end of the analog range and uses the standard formula otherwise.

The definition has two other notable features. One, it signals an error if invoked with zero or a negative number of digital levels — no way to make sense out of such a request. Two, it uses the function fromIntegral to make the divisor compatible with the dividend in the computation of the step size. The function is packaged with some other utilities for numeric computation in a module called NumericUtilities, (provided in the Appendix).

```
¿ HASKELL DEFINITION ?   -- n-way analog-to-digital converter for a <= x < b
¿ HASKELL DEFINITION ?     digitize:: RealFrac num => Int -> num -> num -> num -> Int
¿ HASKELL DEFINITION ?     digitize n a b x                          -- you write this function
¿ HASKELL DEFINITION ?       | xDist < halfStep   = 0                -- watch out for x near a or b
¿ HASKELL DEFINITION ?       | xDist > (analogRangeSize - halfStep)   = nSafe - 1
¿ HASKELL DEFINITION ?       | otherwise   = floor(xDist/dx)
¿ HASKELL DEFINITION ?       where
¿ HASKELL DEFINITION ?       xDist = x - a
¿ HASKELL DEFINITION ?       dx = analogRangeSize/(fromIntegral nSafe)
¿ HASKELL DEFINITION ?       halfStep = dx/2
¿ HASKELL DEFINITION ?       nSafe   | n > 0   = n
¿ HASKELL DEFINITION ?               | otherwise   = error "digitize: zero or negative levels"
¿ HASKELL DEFINITION ?       analogRangeSize = b - a                                          1
```

The function digitize is polymorphic: it can deal with any representation of analog values in the class RealFrac. This includes not only floating point numbers, but also rational numbers. Rational numbers are constructed from a numerator and denominator, both of which are integral numbers. If the numerator and denominator have type Integer, then the rational number has type Rational (short for Ratio Integer). If they have type Int, then the rational number has type Ratio Int. Rational numbers are written as a pair of Haskell integral numbers with a percent sign between them. The graph-plotting function, showGraph, will make use of a rational number denoted in this way.

With the digitizing function understood, the next step is to construct the graph-plotting function. This will be done in steps, from a version that is relatively easy to design, but not very desirable to use, to a version that has more complex formulas, but is more convenient to use.

***Haskell notations for numbers in class* RealFrac**

| *numbers of type* Float *or* Double | | *numbers of type* Rational *or* Ratio Int |
|---|---|---|
| 3.14159 | ~~.1~~ | 3%5 — *three fifths* |
| 0.31415926e+01 | *dangling decimal point not allowed* | 5%3 — *five thirds* |
| 31415926356.0e-10 | | -279%365 — *negative two hundred seventy-nine three hundred sixty-fifths* |
| -3.1415926 | ~~100.~~ | |
| -0.31416e+01 | | |

The graph-plotting function will deliver a string that, when displayed on the screen, will appear as lines containing asterisks to form the curve that represents the graph of the function being plotted. The arguments supplied to the graph-plotting function will include the function to be plotted, the extent of the domain over which to plot the function, and the desired number of digitizing levels to break the range into.

*graph-plotting function* — showGraph
*arguments*
- w — number of digitizing levels for the abscissa (a value of type Int)
- f — function to plot (type num->num, where num is a type in the class RealFrac)
- a — left-hand endpoint of the domain over which to plot the function
- b — right-hand endpoint of the domain over which to plot the function

*result delivered*
- string that will display a curve representing the function-graph $\{f\ x \mid a < x < b\}$

The function will first build a sequence of strings, each to become one line in the result, then apply the intrinsic function unlines to convert this sequence of strings in to one string with newline characters separating the strings in the original sequence.

> **unlines :: [String] -> String**
>
> concatenates all the strings in the argument together and inserts newline character at the end of each
>
> unlines ["IEEE", "Computer"] = "IEEE\nComputer\n"
>
> unlines = concat . map(++"\n")

The string will display the curve with the abscissa running down the screen[1] for w lines in all (one line for each segment in the digitized version of the abscissa). The function will need to choose some appropriate level of digitization for the ordinate. Initially, this will be 20, corresponding to 20 character positions across a line, but it could be any number, as long the printed characters will fit on a line of the printing device. (If they were to wrap around or get lopped off, the graph wouldn't look right.)

The step size in the direction of the abscissa will be $dx = (b - a) / w$, so digital level $k$ corresponds to the segment $a + k*dx < x < a + (k+1)*dx$. The function's value at the centers of these segments will be plotted. This means that the function values must be computed at the set of points

$$\{ a + dx / 2 + k*dx \mid n \in \{0, 1, \ldots, w-1\} \}$$

1. This is not very desirable. The abscissa is normally plotted along the horizontal axis. This is one of the things to be improved in subsequent versions of the showGraph function.

The maximum and minimum of the function values at these points (call them *yMax* and *yMin*) determine the range of values on the ordinate scale. This scale will be digitized into 20 levels by applying digitize to each of the function values.

The sketch of the function showGraph outlines this plan. Try to fill in the details yourself, to make sure you understand how to apply the concepts and formulas presented so far.

**maximum, minimum ::**
**Real num => [num] -> num**

computing the largest or smallest number in a sequence

maximum[5, 9, 2] = 9
minimum[4.7, -1.3,  3.14] = -1.3

```
¿ HASKELL DEFINITION ?    showGraph:: RealFrac num =>
¿ HASKELL DEFINITION ?      Int -> (num->num) -> num -> num -> String
¿ HASKELL DEFINITION ?    showGraph w f a b = unlines graph
¿ HASKELL DEFINITION ?      where
¿ HASKELL DEFINITION ?      graph = [spaces y ++ "*" | y <- ysDigitized]
¿ HASKELL DEFINITION ?      ysDigitized =                -- use the digitize function for this
¿ HASKELL DEFINITION ?        [digitize 20 yMin yMax y | y<-ys]
¿ HASKELL DEFINITION ?      ys = [f x| x<-xs]                                    -- ordinates
¿ HASKELL DEFINITION ?      xs =                     -- centered abscissas (you define xs)
¿ HASKELL DEFINITION ?        [a + dx/2 + fromInt(k)*dx| k<-[0..w-1]]
¿ HASKELL DEFINITION ?      dx =                   -- step size for abscissa (you define dx)
¿ HASKELL DEFINITION ?        (b-a)/fromIntegral(w)
¿ HASKELL DEFINITION ?      yMax = maximum ys
¿ HASKELL DEFINITION ?      yMin = minimum ys
```

fromIntegral *converts* w *to the type of* (b-a)

2

```
HASKELL COMMAND •  putStr(showGraph 20 sin (-2*pi) (2*pi))
HASKELL RESPONSE •              *
HASKELL RESPONSE •                   *
HASKELL RESPONSE •                    *
HASKELL RESPONSE •                   *
HASKELL RESPONSE •              *
HASKELL RESPONSE •       *
HASKELL RESPONSE •  *
HASKELL RESPONSE • *
HASKELL RESPONSE •  *
HASKELL RESPONSE •       *
HASKELL RESPONSE •              *
HASKELL RESPONSE •                   *
HASKELL RESPONSE •                    *
HASKELL RESPONSE •                   *
HASKELL RESPONSE •              *
HASKELL RESPONSE •       *
HASKELL RESPONSE •  *
HASKELL RESPONSE • *
HASKELL RESPONSE •  *
HASKELL RESPONSE •       *
```

**sin :: Floating num =>**
**num-> num**

sin *is an intrinsic function that delivers the (approximate) trigonometric sine of a floating point argument*

**pi :: Floating num => num**

pi *is an intrinsic variable whose value approximates the ratio of the circumference of a circle to its diameter*

In the definition of showGraph, the variable graph is a sequence of strings, one string for each line that will appear in the display. Each of these lines is a sequence of spaces (delivered by a function, spaces — see SequenceUtilities (Appendix) followed by an asterisk. The spaces shift the asterisk further to the right for larger function values, and the overall effect is a curve showing the behavior of the function, as shown in the following Haskell command and response.

It's a little disorienting to see the curve running down the page. Normally the abscissa is plotted in the horizontal direction. The next version of showGraph corrects this situation.

Think of the display of the graph as a table of rows and columns of characters. The rows go across the page and the columns go up and down. Each row has 20 characters, since that is the number of digitized levels in the ordinate, and each column has w characters, since w specifies the number of digitized levels in the abscissa.

To display the graph in the usual orientation (horizontal axis for the abscissa), the last column of the table (that is, the right-most column) needs to become the top row, the next-to-last column the second row, and so on. This is known as a transposition of rows and columns in the table. A function called transpose in the SequenceUtilities module (Appendix) that does this operation.

Well … not quite. The function transpose actually makes the left-most column the top row and the right-most column the bottom row, rather than the other way around.[1] This can be fixed by reversing the order of the strings that plot the abscissas before feeding this sequence of strings to the unlines function.

However, there is a slight complication that needs to addressed before the transpose function will work properly in this application. The complication is that the strings in the graph variable are not full rows. They don't have all 20 characters in them. Instead, they have just enough spaces, followed by an asterisk, to plot a point on the graph in the right position.

For transpose to work as intended, the rows must be full, 20-column units. So, the formula for a row must append enough spaces on the end to fill it out to 20 columns. Try to put the proper row-formula in the following version of showGraph.

```
¿ HASKELL DEFINITION ?    showGraph:: RealFrac num =>
¿ HASKELL DEFINITION ?      Int -> (num->num) -> num -> num -> String
¿ HASKELL DEFINITION ?    showGraph w f a b = (unlines . reverse . transpose) graph
¿ HASKELL DEFINITION ?      where
¿ HASKELL DEFINITION ?      graph                          -- you fill in the formula for graph
¿ HASKELL DEFINITION ?        = [spaces y ++ "*" ++ spaces(20 - (y+1))| y <- ysDigitized]
¿ HASKELL DEFINITION ?      ysDigitized = [digitize 20 yMin yMax y| y<-ys]
¿ HASKELL DEFINITION ?      ys = [f x| x<-xs]
¿ HASKELL DEFINITION ?      xs = [a + dx/2 + fromIntegral(k)*dx| k<-[0..w-1]]
¿ HASKELL DEFINITION ?      dx = (b-a)/fromIntegral(w)
```

1. The function transpose is designed to work on matrices. According to the usual conventions in mathematics, the transpose of a matrix makes the left-most column into the top row, the second column (from the left) into the second row, and so on.

¿ *HASKELL DEFINITION ?* yMax = maximum ys

¿ *HASKELL DEFINITION ?* yMin = minimum ys 3

With this change, the showGraph function displays the graph in the usual orientation (abscissa running horizontally).

```
HASKELL COMMAND •  putStr(showGraph 20 sin (-2*pi) (2*pi))
HASKELL RESPONSE •   *         *
HASKELL RESPONSE •  * *       * *
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE • *   *     *   *
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •      *   *     *   *
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •
HASKELL RESPONSE •       * *       * *
HASKELL RESPONSE •        *         *
```

Wait a minute! Why is the graph all squeezed up?

There are two factors involved in this phenomenon. One is that the program exercises no discretion about how many levels to use in digitizing the ordinate. It just picks 20 levels, no matter what. So, some graphs will look squeezed up, some spread out, depending on scale.

This can be fixed by scaling the ordinate to match the abscissa, so that a unit moved in the vertical direction on the plotting device will correspond to about the same range of numbers as a unit moved in the horizontal direction. Another way to look at this is to choose the scaling factor so that a segment in the range of the abscissa that corresponds to one digitization level has the same length as a digitization segment in the range of the ordinate. In arithmetic terms, the following proportions need to be approximated:

$$height \,/\, w = (yMax - yMin) \,/\, (b - a)$$

where *height* is the number of digitizing levels in the vertical (ordinate) direction.

The other factor is that the resolution of the printer in the vertical direction is not the same as the resolution in the horizontal direction. Typically a movement on the printer in the vertical direction is about twice as far as a movement in the horizontal direction. The exact ratio depends on the printer, but a ratio of about five to three is typical. So, to get the proportions right, horizontal units need to be adjusted by a factor of three-fifths to make them comparable to vertical units.

Combining this aspect ratio of the horizontal and vertical resolutions of the printer with the maintenance of the above scaling proportions leads to the following formula for the number of digitizing levels in the vertical direction:

$$\textit{height} = \text{nearest integer to } w * \frac{3}{5} * (yMax - yMin) / (b - a)$$

The final version of showGraph is packaged in a module for use in other scripts. The module assumes that the function digitize can be imported from a module called NumericUtilities and that the functions spaces and transpose can be imported from a module called SequenceUtilities.

Try to use the above formulas to fill in the details of the function showGraph. To compute the nearest integer to a fractional number, apply the intrinsic function round. Note that the transpose function has been packaged in the SequenceUtilities module, and the digitize function has been packaged in the NumericUtilities module. Both of these modules are contained in the Appendix.

```
¿ HASKELL DEFINITION ?   module PlotUtilities
¿ HASKELL DEFINITION ?     (showGraph)
¿ HASKELL DEFINITION ?     where
¿ HASKELL DEFINITION ?     import SequenceUtilities(transpose)
¿ HASKELL DEFINITION ?     import NumericUtilities(digitize)
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?     showGraph:: RealFrac num =>
¿ HASKELL DEFINITION ?       Int -> (num->num) -> num -> num -> String
¿ HASKELL DEFINITION ?     showGraph w f a b = (unlines . reverse . transpose) graph
¿ HASKELL DEFINITION ?       where
¿ HASKELL DEFINITION ?       graph =                                   -- you define graph
¿ HASKELL DEFINITION ?         [spaces y ++ "*" ++ spaces(height-1 - y)| y <- ysDigitized]
¿ HASKELL DEFINITION ?       ysDigitized = [digitize height yMin yMax y| y<-ys]
¿ HASKELL DEFINITION ?       height =                                  -- you define height
¿ HASKELL DEFINITION ?         max 1 (round(fromIntegral(w)*aspect*(yMax - yMin)/(b - a)))
¿ HASKELL DEFINITION ?       ys = [f x| x<-xs]
¿ HASKELL DEFINITION ?       xs = [a + dx/2 + fromIntegral(k)*dx| k<-[0..w-1]]
¿ HASKELL DEFINITION ?       dx = (b-a)/fromIntegral(w)
¿ HASKELL DEFINITION ?       yMax = maximum ys
¿ HASKELL DEFINITION ?       yMin = minimum ys
¿ HASKELL DEFINITION ?       aspect = fromRational(3%5)
```

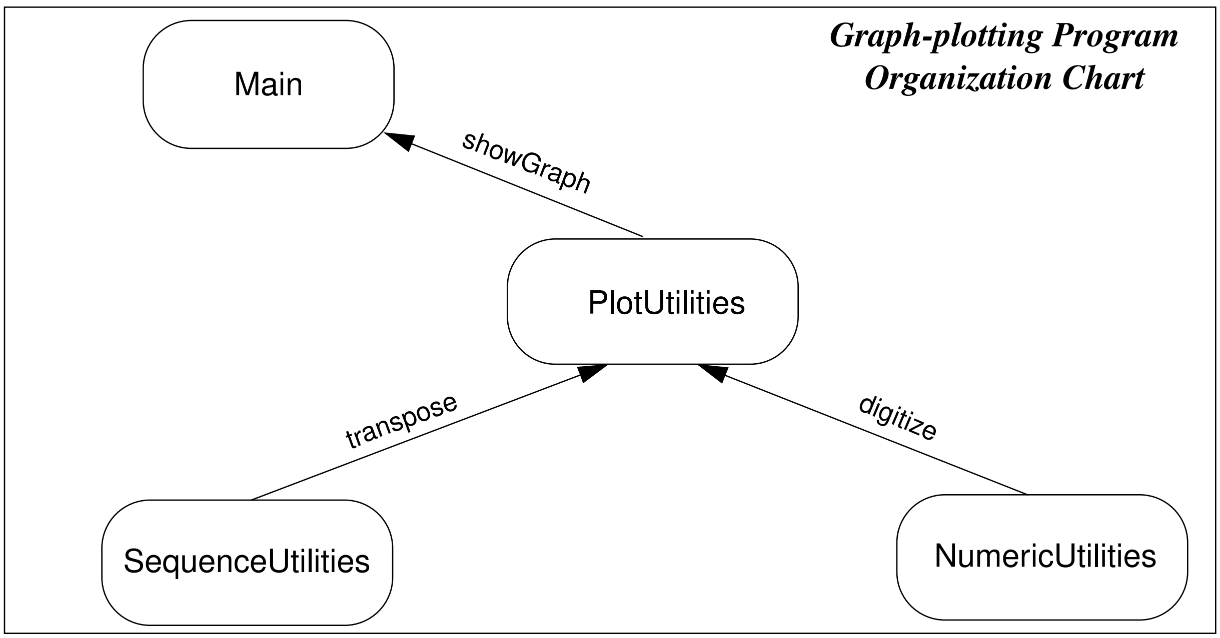
4

The following command applies showGraph in the usual way.

```
HASKELL DEFINITION •   import PlotUtilities(showGraph)
HASKELL COMMAND •      putStr(showGraph 20 sin (-2*pi) (2*pi))
HASKELL RESPONSE •     *****       *****
HASKELL RESPONSE •          *****       *****
```

Whoops! Poor resolution in the vertical direction. Doubling the resolution gives a better picture.

```
HASKELL COMMAND •      putStr(showGraph 40 sin (-2*pi) (2*pi))
HASKELL RESPONSE •       ******                   ******
HASKELL RESPONSE •     **      **           **      **
HASKELL RESPONSE •               **      **           **      **
HASKELL RESPONSE •                 ******                   ******
```




## *Review Questions*

1 The Haskell class Fractional includes
   a integral, real, and complex numbers ← *correct answer*
   b numbers between zero and one, but not numbers bigger than one
   c both floating point and rational numbers
   d the Mandelbrot set

2 The mantissa of a floating point number determines
   a where the decimal point goes
   b the range of the number and its sign
   c the magnitude and precision of the number
   d the sign of the number and the digits in its decimal numeral ← *correct answer*

3 The exponent of a floating point number determines
  a where the decimal point goes ← *correct answer*
  b the range of the number and its sign
  c the magnitude and precision of the number
  d the sign of the number and the digits in its decimal numeral

4 The following denote floating point numbers as the should appear in a Haskell script
  a 1.89533e+25, 18.01528974, 1.05522e-24, +27.0 ← *incorrect — can't put plus sign in front of number*
  b $1.89533 \times 10^{25}$, 18.01528974, $1.05522 \times 10^{-24}$, -27.0
  c 1.89533e+25, 18.01528974, 1.05522e-24, -27.0 ← *correct answer*
  d all of the above

5 Analog to digital conversion converts a number
  a from a set containing a great many numbers to a number from a much smaller set ← *correct answer*
  b to zero or one
  c to a pattern of zeros and ones
  d by a digital analogy process

6 Which of the following formulas would useful for analog to digital conversion?
  a `floor((x - a)/dx)`
  b `floor(n*(x - a)/(b - a))`
  c `floor . (/ dx) . (+(- a))`
  d all of the above ← *correct answer*

7 Numbers of type `Rational` in Haskell scripts are
  a compatible with floating point numbers in arithmetic operations
  b constructed from two integers by putting a percent-sign between them ← *correct answer*
  c especially useful when precision is not the most important factor
  d all of the above

# *Patterns as Formal Parameters* 21

When you know something about the structure of an argument that may be supplied to a function, you can take advantage of that knowledge to make the definition more concise and easier to understand. For example, suppose you are writing a function whose argument will be a two-tuple of numbers, and the function is supposed to deliver the sum of those numbers. You could write the definition as follows.

```
HASKELL DEFINITION •   sumPair :: Num num => (num, num) -> num
HASKELL DEFINITION •   sumPair (x, y) = x + y
```

The formal parameter in this case is a two-tuple pattern. When the function is used in a formula, it will be supplied with a two-tuple of numbers as an argument. At that point, the first component of the tuple argument gets associated with the first component of the tuple-pattern in the definition (that is, x), and the second component of the tuple argument gets associated with the second component of the tuple-pattern (that is, y).

```
HASKELL COMMAND •    sumPair(12, 25)      — matches x in definition with 12, y with 25
HASKELL RESPONSE •   37                   — delivers 12 + 25
```

This idea can also be used with arguments that are sequences. For example, the following function expects its argument to be a sequence of two strings, and it returns a string containing the first character in the first string and the last character in the second string.

```
HASKELL DEFINITION •   firstAndLast :: [String] -> String
HASKELL DEFINITION •   firstAndLast [xs, ys] = [head xs] ++ [last ys]
```

This function could be generalized to handle arguments with other sequence-patterns. Values to be delivered for other patterns are simply written in separate equations. The following definition would cover three cases: (1) an argument with two elements, as above, (2) an argument with one element, and (3) an argument with no elements.

```
HASKELL DEFINITION •   firstAndLast :: [String] -> String
HASKELL DEFINITION •   firstAndLast [xs, ys] = [head xs] ++ [last ys]
HASKELL DEFINITION •   firstAndLast [xs] = [head xs] ++ [last xs]
HASKELL DEFINITION •   firstAndLast [ ] = [ ]
```

This amounts to a function with three separate cases in its definition. The appropriate case is selected by matching the supplied argument against the patterns in the defining equations and choosing the defining equation that matches. If no pattern matches the supplied argument, the function is not defined for that argument. The preceding definition of firstAndLast does not define the function on sequences with three or more elements.

To define firstAndLast on sequences with any number of elements, a pattern involving the sequence constructor can be used. The **sequence constructor** is an operator denoted by the colon (:) that inserts a new element at the beginning of an existing sequence.

```
x : xs  =  [x] ++ xs
```

Of course, you already know how to insert an element at the beginning of an existing sequence by using the append operator (++). Unfortunately, however, the append operator is not included in the class of operators that can be used to form patterns in formal parameters. Operators in this class are known as constructors, and it just happens that the colon operator is one of those, but the plus-plus operator isn't.

Using the sequence constructor, the definition of firstAndLast can be extended to deal with all finite sequences:

```
HASKELL DEFINITION •   firstAndLast :: [String] -> String
HASKELL DEFINITION •   firstAndLast (xs : yss) = [head xs] ++ [last(last(xs : yss))]
HASKELL DEFINITION •   firstAndLast [ ] = [ ]
```

In this definition, the first equation will be selected to deliver the value if the supplied argument has one or more elements because the pattern (xs : yss) denotes a sequences that contains at least the element xs. If the supplied argument has no elements, then the second equation will be selected.

```
HASKELL COMMAND •    firstAndLast ["A", "few", "words"]        — selects first equation
HASKELL RESPONSE •   As
HASKELL COMMAND •    firstAndLast["Only", "two"]               — selects first equation
HASKELL RESPONSE •   Oo
HASKELL COMMAND •    firstAndLast["one"]                       — selects first equation
HASKELL RESPONSE •   oe
HASKELL COMMAND •    firstAndLast [ ]                          — selects second equation
HASKELL RESPONSE •   [ ]
```

Many definitions use patterns involving the sequence constructor (:) because it often happens that a different formula applies when an argument is non-empty than when the argument is empty.[1] Of course, you could always write the definition using guards:

```
HASKELL DEFINITION •   firstAndLast :: [String] -> String
HASKELL DEFINITION •   firstAndLast xss
HASKELL DEFINITION •      | null  xss     = [ ]
HASKELL DEFINITION •      | otherwise   = [head(head  xss)] ++ [last(last  xss)]
```

**head, last :: [a] -> a** — *intrinsic functions*
**tail :: [a] -> [a]**
head([$x$] ++ $xs$) = $x$    tail([$x$] ++ $xs$) = $xs$
last = head . reverse

But, the pattern-matching form of the definition has the advantage of attaching names to the components of the sequence that can be used directly in the definition, rather than having to apply head or tail to extract them. For example, in the pattern-matching form of the definition of firstAndLast, the first component of the argument sequence in the non-empty case as associated with the name xs. So, it can be used in the definition: head xs, rather than the more complicated head(head xss) required in the definition that does not rely on pattern-matching.

1. In the firstAndLast function, for example, the an empty argument presents a special case because there are no strings from which to extract first and last elements.

## *Review Questions*

1 The formula (x : xs) is equivalent to
- a x ++ xs
- b [x] ++ xs ← *correct answer*
- c [x] ++ [xs]
- d all of the above

2 The definition

HASKELL DEFINITION • f(x : xs) = g x xs
HASKELL DEFINITION • f [ ] = h

- a defines h in terms of g
- b defines f for arguments that are either empty or non-empty sequences ← *correct answer*
- c will not work if xs is the empty sequence
- d all of the above

3 The definition

HASKELL DEFINITION • f(x : xs) = g x xs

is equivalent to
- a f xs | null xs = g x xs
- b f xs = g x xs || h
- c f xs | not(null xs) = g (head x) (tail xs) ← *correct answer*
- d f x xs = g(x : xs)

4 Which of the following defines a function of type ([Char], Char) -> [Char] ?
- a f( (x : xs), 'x' ) = [x] ++ reverse xs ++ ['x'] ← *correct answer*
- b f( x , y : ys ) = [ ] ++ reverse ys ++ [x]
- c f( (xs : 'x'), x ) = [x] ++ reverse xs ++ ['x']
- d all of the above

5 Which of the following formulas delivers every third element of the sequence xs?
- a foldr drop [ ] xs
- b [ foldr drop [ ] suffix | suffix <- iterate (drop 3) xs ]
- c [ x | (x : suffix) <- takeWhile (/= [ ]) (iterate (drop 3) (drop 2 xs)) ] ← *correct answer*
- d takeWhile (/= [ ]) (iterate (take 3) xs)

# *Recursion* 22

You have learned to use several patterns of computation that involve repetition of one sort or another: mapping (applying the same function to each element in a sequence), folding (collapsing all the elements of a sequence into one by inserting a binary operation between each adjacent pair), iterating (applying a function to its own output, repeatedly), filtering (forming a new sequence from the elements of an existing one that pass a certain criterion), and extracting a prefix or suffix of a sequence. Most important computations can be described using just these patterns of repetition. But not all.

Some computations require other patterns of repetition. In fact, there is no finite collection of patterns that cover all of the possibilities. For this reason, general purpose programming languages must provide facilities to permit the specification of arbitrary patterns of repetition. In Haskell, recursion provides this capability.

A definition that contains a formula that refers to the term being defined is called a **recursive formula**. All of the patterns of repetition that you have seen can be described with such formulas.

Take iteration, for example. The iterate function constructs a sequence in which each successive element is the value delivered by applying a given function to the previous element. It is an intrinsic function, of course, but if it weren't, the following equation would define it.

*HASKELL DEFINITION* • `iterate f x = [ x ] ++ iterate f (f x)`

What does this mean? It means that the value iterate delivers will be a sequence whose first element is the same as the second argument supplied to iterate and whose subsequent elements can be computed by applying iterate to different arguments. Well … not completely different. The first argument is the same as the first argument originally supplied to iterate. The second argument is different, however. What was x before is now (f x).

Therefore, the first element of the value delivered by the subformula iterate f (f x) will be (f x). This value becomes the second element in the sequence delivered by iterate f x. What about the third element? The third element will be the second element delivered by the subformula iterate f (f x).

To see what this value is, just re-apply the definition of iterate:

`iterate f (f x) = [(f x)] ++ iterate f (f (f x))`

The second element in this sequence is the first element in the sequence delivered by the subformula iterate f (f (f x)), and that value is (f (f x)), as you can see from the definition of iterate.

And so on. This is how recursion works.

Look at another example: the function foldr, defined via recursion:

*HASKELL DEFINITION* • `foldr op z (x : xs) = op x (foldr op z xs)`
*HASKELL DEFINITION* • `foldr op z [ ] = z`

You can see the pattern of repetition that this definition leads to by applying the definition to the formula foldr (+) 0 [1, 2, 3].

```
foldr (+) 0 [1, 2, 3]   = (+) 1 (foldr (+) 0 [2, 3])          — according to the definition of foldr
                        = 1 +  (foldr (+) 0 [2, 3])           — switching to operator notation for (+)
                        = 1 + ((+) 2  (foldr (+) 0 [3])       — applying the definition of foldr again
                        = 1 +  (2 + (foldr (+) 0 [3]))        — switching to operator notation for (+)
                        = 1  + (2 + ((+) 3 (foldr (+) 0 [ ])))  — applying the definition again
                        = 1  + (2 + (3 + (foldr (+) 0 [ ])))  — switching to operator notation for (+)
                        = 1  + (2 + (3 + 0))    — applying the definition again (empty-case this time)
```

This is the operational view of recursion — how it works. Generally, it's not a good idea to worry about how recursion works when you are using it to specify computations. What you should concern yourself with is making sure the equations you write establish correct relationships among the terms you are defining.

Try to use recursion to define the take function. The trick is to make the defining formula push the computation one step further along (and to make sure your equations specify correct relationships).

```
¿ HASKELL DEFINITION ?   take  n  (x : xs)                       — you take a stab at the definition
¿ HASKELL DEFINITION ?      | n > 0       = [x] + take (n - 1) xs
¿ HASKELL DEFINITION ?      | n == 0      = [ ]
¿ HASKELL DEFINITION ?      otherwise   = error("take (" ++ show  n ++ ") not allowed")
¿ HASKELL DEFINITION ?   take  n  [ ] = [ ]                      — don't forget this case
```

So much for using recursion to define what you already understand. Now comes the time to try it on a new problem.

Suppose you have a sequence of strings that occur in more-or-less random order and you want to build a sequence containing the same elements, but arranged alphabetical order. This is known as sorting. The need for sorting occurs so frequently that it accounts for a significant percentage of the total computation that takes place in businesses worldwide, every day. It is one of the most heavily studied computations in computing.

There are lots of ways to approach the sorting problem. If you know something about the way the elements of the sequence are likely to be arranged (that is, if the arrangement is not uniformly random, but tends to follow certain patterns), then you may be able to find specialized methods that do the job very quickly. Similarly if you know something about the elements themselves, such as that they are all three-letter strings, then you may be able to do something clever. Usually, however, you won't have any specialized information. The sorting method discussed in this chapter is, on the average, the fastest known way[1] to sort sequences of elements when you don't know anything about them except how to compare pairs of elements to see which order they should go in.

Fortunately, it is not only the fastest known method, it is also one of the easiest to understand. It was originally discovered by C. A. R. Hoare in the early days of computing. He called it quicksort, and it goes like this: Compare each element in the sequence to the first element. Pile up the elements that should precede it in one pile and pile up the elements that should follow it in another pile. Then apply the sorting method to both piles (this is where the recursion comes in). When you

1. There are all sorts of tricks that can be applied to tweak the details and get the job done faster, but all of these tricks leave the basic method, the one discussed in this chapter, in place.

are finished with that, build a sequence that (1) begins with the elements from first pile (now that they have been sorted), (2) then includes the first element of the original sequence, and (3) ends with the elements from the second pile (which have also been sorted at this point).

Try your hand at expressing the quick-sort computation in Haskell.

```
¿ HASKELL DEFINITION ?   quicksort (firstx : xs) =                 — you try to define quicksort
¿ HASKELL DEFINITION ?      quicksort[x | x <- xs, x < firstx] ++ [firstx] ++
¿ HASKELL DEFINITION ?      quicksort[x | x <- xs, not(x < firstx)]
¿ HASKELL DEFINITION ?   quicksort [ ] = [ ]
   HASKELL COMMAND •     quicksort["Billy", "Sue", "Tom", "Rita"]
  HASKELL RESPONSE •     ["Billy", "Rita", "Sue", "Tom"]                  — works on strings
   HASKELL COMMAND •     quicksort[32, 5280, 12, 8]
  HASKELL RESPONSE •     [8, 12, 32, 5280]                            — works on numbers, too
   HASKELL COMMAND •     quicksort[129.92, -12.47, 59.99, 19.95]
  HASKELL RESPONSE •     [-12.47, 19.95, 59.99, 129.92]
   HASKELL COMMAND •     quicksort["Poe", "cummings", "Whitman", "Seuss", "Dylan"]
  HASKELL RESPONSE •     ["Dylan", "Poe", "Seuss", "Whitman", "cummings"]       — whoops!
```

As written, quicksort puts numbers in increasing order and puts strings in alphabetical order. But, it seems to have some sort of lapse in the last of the preceding examples. It puts "cummings" last, when it should be first, going in alphabetical order.

The problem here is that quicksort is using the intrinsic comparison operation (<), and this operation arranges strings in the order determined by the ord function, applied individually to characters in the strings. The ord function places capital letters prior to lower case letters, so "cummings" is last because it starts with a lower case letter.

This kind of problem applies to many kinds of things you might want to sort. For example, if you had a sequence of tuples containing names, addresses, and phone numbers of a group of people, you might want to sort them by name, or by phone number, or by city. The built in comparison operation (<), no matter how it might be defined on tuples, could not handle all of these cases.

What the quicksort function needs is another argument. It needs to be parameterized with respect to the comparison operation. Then, an invocation could supply a comparison operation that is appropriate for the desired ordering.

A version of quicksort revised in this way is easy to construct from the preceding definition. Try to do it on your own.

```
¿ HASKELL DEFINITION ?   quicksortWith precedes (firstx : xs)        — you define quicksortWith
¿ HASKELL DEFINITION ?      quicksortWith precedes [x | x <- xs, precedes x firstx] ++
¿ HASKELL DEFINITION ?      [firstx] ++
¿ HASKELL DEFINITION ?      quicksortWith precedes [x | x <- xs, not(precedes x firstx)]
¿ HASKELL DEFINITION ?   quicksortWith precedes [ ] = [ ]
```

Now, if the intrinsic comparison operation (<) is supplied as the first argument of quicksortWith, it will work as quicksort did before.

```
HASKELL COMMAND •   quicksortWith (<) ["Poe", "cummings", "Whitman", "Seuss", "Dylan"]
HASKELL RESPONSE •  ["Dylan", "Poe", "Seuss", "Whitman", "cummings"]
```

However, if a special operation is provided to do a better job of alphabetic comparison, then quicksort can deliver an alphabetical arrangement that is not subject to the whims of ord.

```
HASKELL DEFINITION •   import Char  -- get access to toLower function
HASKELL DEFINITION •   precedesAlphabetically  x  y
HASKELL DEFINITION •      | xLower == yLower     =  x < y
HASKELL DEFINITION •      | otherwise                    =  xLower < yLower
HASKELL DEFINITION •      where
HASKELL DEFINITION •      xLower = map toLower  x
HASKELL DEFINITION •      yLower = map toLower  y
¿ HASKELL COMMAND ?   quicksortWith precedesAlphabetically        — you write the invocation
¿ HASKELL COMMAND ?      ["Poe", "cummings", "Whitman", "Seuss", "Dylan"]
HASKELL RESPONSE •   ["cummings", "Dylan", "Poe", "Seuss", "Whitman"]
```

The new version of quicksort is a general purpose sorting method for sequences. It can be applied to any kind of sequence, as long as a comparison operation is supplied to compare the elements of the sequence.

## *Review Questions*

1 Which of the following defines a function that delivers the same results as the intrinsic function reverse?
- a rev(x : xs) = xs ++ [x]
  rev [ ] = [ ]
- b rev(xs : x) = x : xs
  rev [ ] = [ ]
- c rev(x : xs) = rev xs ++ [x] ← *correct answer*
  rev [ ] = [ ]
- d none of the above

2 Which of the following defines a function that would rearrange a sequence of numbers to put it in decreasing numeric order?
- a sortDecreasing = quickSortWith (>) ← *correct answer*
- b sortDecreasing = quickSortWith (>) [18.01528974, 1.89533e+25, 1.05522e-24, 27.0]
- c sortDecreasing = quickSortWith (>) numbers
- d all of the above

3 The following function

```
HASKELL DEFINITION •   sorta(x : xs) = insert x (sorta xs)
HASKELL DEFINITION •   sorta [ ] = [ ]
HASKELL DEFINITION •   insert a (x : xs)
HASKELL DEFINITION •     | a <= x        = [a, x] ++ xs
HASKELL DEFINITION •     | otherwise  = [x] ++ (insert a xs)
HASKELL DEFINITION •   insert a [ ] = [a]
```

- a delivers the same results as quicksort
- b delivers the same results as quicksortWith (<)
- c both of the above ← *correct answer*
- d neither of the above

# Ifs, Lets and Unlimited Interactive Input 23

The interactive programs described up to this point have had a rigid structure. They all performed a fixed number of input/output directives. In each case, the exact number of input/output directives had to be known before the script was written. This is fine as far as it goes, but what do you do when you cannot predict in advance how many input items there might be?

For example, suppose you want to write a script that will ask the person at the keyboard in enter a sequence of names, any number of them, and finally enter some signal string like "no more names", to terminate the input process. Then, the program is to display something based on the names entered, such as displaying the names in alphabetical order (using the quicksortWith function, which has been packaged in the SequenceUtilities module from the Appendix). In a case like this, you cannot know in advance how many input directives there will be. So, you cannot use a do-expression made up of a simple list of input/output directives.

The solution to the problem is to use a recursive formulation of the input function to continue the process as long as necessary and to select an alternative formulation, without the recursion, when the special signal (e.g., "no more names") is entered. The following script does this. It uses two new kinds of expressions: let expressions and conditional expressions (if-then-else). Take a look at the script, and try to follow the logic. The new constructs are explained in detail in the text following the script.

```
HASKELL DEFINITION •  import Char(toLower)
HASKELL DEFINITION •  import SequenceUtilities(quicksortWith)
HASKELL DEFINITION •
HASKELL DEFINITION •  main =
HASKELL DEFINITION •    do
HASKELL DEFINITION •      names <- getNames
HASKELL DEFINITION •      do
HASKELL DEFINITION •        let sortedNames = quicksortWith namePrecedes names
HASKELL DEFINITION •        putStr(unlines sortedNames)
HASKELL DEFINITION •
HASKELL DEFINITION •  getNames =
HASKELL DEFINITION •    do
HASKELL DEFINITION •      name <- getName
HASKELL DEFINITION •      if name == "no more names"
HASKELL DEFINITION •        then return [ ]
HASKELL DEFINITION •        else
HASKELL DEFINITION •          do
HASKELL DEFINITION •            names <- getNames
HASKELL DEFINITION •            return([name] ++ names)
HASKELL DEFINITION •
HASKELL DEFINITION •  getName =
HASKELL DEFINITION •    do
HASKELL DEFINITION •      putStr "Enter name (or \"no more names\" to terminate): "
HASKELL DEFINITION •      name <- getLine
```

## ANSWERS — WORK OUT P.120Q BEFORE READING THIS PAGE

```
HASKELL DEFINITION •          return name
HASKELL DEFINITION •
HASKELL DEFINITION •   namePrecedes name1 name2 = precedesAlphabetically lnf1 lnf2
HASKELL DEFINITION •      where
HASKELL DEFINITION •      lnf1 = lastNameFirst name1
HASKELL DEFINITION •      lnf2 = lastNameFirst name2
HASKELL DEFINITION •
HASKELL DEFINITION •   lastNameFirst name =
HASKELL DEFINITION •      dropWhile (== ' ') separatorThenLastName ++ " " ++ firstName
HASKELL DEFINITION •      where
HASKELL DEFINITION •      (firstName, separatorThenLastName) = break (== ' ') name
HASKELL DEFINITION •
HASKELL DEFINITION •   precedesAlphabetically :: String -> String -> Bool
HASKELL DEFINITION •   precedesAlphabetically  x  y
HASKELL DEFINITION •      | xLower == yLower     =  x < y
HASKELL DEFINITION •      | otherwise            =  xLower < yLower
HASKELL DEFINITION •      where
HASKELL DEFINITION •      xLower = map toLower  x
HASKELL DEFINITION •      yLower = map toLower  y
```

Directives in a do-expression have a different nature from operations in ordinary formulas. One difference is that the do-expression imposes a sequence on the directives. Another is that variables used to stand for data retrieved from input directives are accessible only in subsequent directives within the do-expression. For these reasons, the where clauses and guarded formulas that you have been using to define functions do not fit into the realm of do-expressions.

Instead, two other notations are used for this purpose: the let expression serves the role of the where clause and the conditional expression (if-then-else) provides a way to select alternative routes through the sequence of input/output directives, much like guarded formulas provided a way to select alternative values for ordinary functions.

A let expression may appear at the beginning of a do-expression to give names to values to be used later in the do-expression. The let expression may contain any number of definitions, each of which associates a name with a value. These appear as equations following the let keyword, one equation per line and indented properly to observe the offsides rule for grouping. Variables defined in let expressions can be used at any subsequent point in the do-expression containing them, but they are not accessible outside that do-expression.

A conditional expression provides a way to select between two alternative sequences of input/output commands. It begins with the keyword if, which is followed by a formula that delivers a Boolean value (True or False). Following the Boolean formula is the keyword then and a sequence of input/output directives. Fnally, the keyword else followed by an alternative sequence of input/outpout directives completes the conditional expression. When the Boolean formula delivers the value True, the computation proceeds with the input/output commands in the then-branch of the conditional expression; otherwise, it proceeds with those in the else-branch.

Take another look at the function getNames in the script. This is the function that has uses recursion to allow the sequence of input/output directives to continue until the termination signal is entered, no matter how many names are entered before that point. The key step occurs in the conditional expression. After retrieving a name from the keyboard, getNames tests it in the Boolean formula following the if keyword in the conditional expression. If the termination string "no more names" was entered, then getNames returns the empty list. Otherwise it returns a sequence beginning with the name retrieved and followed by all the rest of the names entered (as retrieved by the recursive invocation of getNames). In this way, getNames builds a sequence of names from the lines entered at the keyboard.

The rest of the script is composed from bits and pieces that you've seen before. The only other new element is the break function. This is an intrinsic function that splits a given sequence into two parts, breaking it at the first point in the sequence where an element occurs that passes a given test. The sequence is supplied as the second argument of break, and the test is supplied as the first argument of break in the form of a function that delivers a Boolean value when applied to an element of the sequence.

```
break :: (a -> Bool) -> [a] -> ([a],[a])
break breakTest xs =
        (takeWhile (not . breakTest) xs,
         dropWhile (not . breakTest) xs)
```

# Algebraic Types 24

Up to this point, all of the Haskell formulas you have seen or written have dealt with types that are intrinsic in the language: characters, Boolean values, and numbers of various kinds, plus sequences and tuples built from these types, and functions with arguments and values in these domains, etc. This system of types provides a great many ways to represent information.

Some classes of computing problems, however, deal with information that is clumsy to describe in terms of Haskell's intrinsic types. For such problems, it is more effective to be able to design your own types, then write functions making use of those types. Haskell provides a way to do this.

In addition to making it more convenient to describe some computations, types defined by software designers also provide an important measure of safety. The type checking mechanisms in Haskell systems are put to work checking for consistent usage of these newly defined types. Since they cannot mix in unanticipated ways with other types, these consistency checks often prevent subtle and hard-to-find defects from slipping into your definitions.

Suppose, for example, you were creating some software that needed to deal with the primary colors red, yellow, and blue. You could define a data type to represent these colors and use it wherever your program needed to record a color:

```
HASKELL DEFINITION •   data Color = Red | Yellow | Blue
```

This definition of the type Color names the three values the Color can take: Red, Yellow, and Blue. These values are known as the **constructors** of the type, and they are listed in the definition, one after another, separated by vertical bars.[1] Constructor names, like data types, must begin with capital letters.

To take the example a bit further, suppose your software needed to deal with two kinds of geometric figures: circles and rectangles. In particular, the software needs to record the dimensions for each such figure and its color. The following definition would provide an appropriate type for this application:

```
HASKELL DEFINITION •   data Figure =
HASKELL DEFINITION •      Circle Color Double | Rectangle Color Double Double
```

This data type specifies two **fields** for the value that Circle constructs (a field of type Color, to record the color of the Circle, and a field of type Double, to record its radius) and three fields for Rectangle (for color, length, and width). The script could use the Figure data type to define variables.

```
HASKELL DEFINITION •   circle = Circle Red 1
HASKELL DEFINITION •   rectangle = Rectangle Blue 5 2.5
HASKELL DEFINITION •   otherCircle = Circle Yellow pi
```

The above definitions define three variables of type Figure: two circles (a red one with unit radius and a yellow one with radius π) and a blue rectangle twice as long as it is wide.

1. This vertical bar is the same one used in list comprehensions, but in the context of data definitions, you should read it as "or." A value of type Color, for example, is either Red or Yellow or Blue.

When you define data types, you will normally want them to inherit certain intrinsic operations, such as equality tests (==, /=) and the show operator, which converts values to strings, so that they can be displayed on the screen or written to files. To accomplish this, attach a deriving clause to the definition that names the classes whose operators the type is to inherit.

```
HASKELL DEFINITION •   data Color =
HASKELL DEFINITION •         Red | Yellow | Blue
HASKELL DEFINITION •         deriving (Eq, Ord, Enum, Show)
HASKELL DEFINITION •
HASKELL DEFINITION •   data Figure =
HASKELL DEFINITION •         Circle Color Double | Rectangle Color Double Double
HASKELL DEFINITION •         deriving (Eq, Show)
```

With the above inheritance characteristics, equality and show operators can be applied to values of either Color or Figure type. In addition, order operators (<, >, etc.) can be applied to Color data, and sequences can be constructed over ranges of Color values.

> **show :: Text a => a -> String**
>
> show 2 = "2"
>
> show (3+7) = "10"
>
> show "xyz" = " \"xyz\" "
>
> show ’x’ = " ’x’ "
>
> - show delivers a string that would denote, in a script, the value of its argument
> - useful primarily in putting together strings for output to the screen or files

```
HASKELL COMMAND •      Red < Yellow
¿ HASKELL RESPONSE ?   True
HASKELL COMMAND •      [Red .. Blue]
¿ HASKELL RESPONSE ?   [Red, Yellow, Blue]
HASKELL COMMAND •      Circle Red 1  ==  Circle Red 2
¿ HASKELL RESPONSE ?   False
HASKELL COMMAND •      show(Rectangle Blue 5 2.5)
HASKELL RESPONSE •     "Rectangle Blue 5.0 2.5"
HASKELL COMMAND •      [Circle Red 1 .. Circle Blue 2]
HASKELL RESPONSE •     ERROR: Figure is not an instance of class "Enum"
```

The last command makes no sense because the type Figure is not in the Enum class. The deriving clause for Figure could not include the Enum class because only **enumeration types** (that is, types whose constructors have no fields) can be in that class.

The fields in type Figure have specific types (Color, Double). But, this need not always be the case. A field can polymorphic. For example, a script might want to use different kinds of numbers to represent the dimensions of circles and rectangles — Double in one part of the script, Integer in another, and perhaps Rational in a third part of the script.

To define polymorphic types, a type parameter (or several type parameters) can be included in the definition:

```
HASKELL DEFINITION •   data (Real realNumber) =>
HASKELL DEFINITION •      Figure realNumber =
HASKELL DEFINITION •         Circle Color realNumber |
HASKELL DEFINITION •         Rectangle Color realNumber realNumber
HASKELL DEFINITION •      deriving (Eq, Show)
```

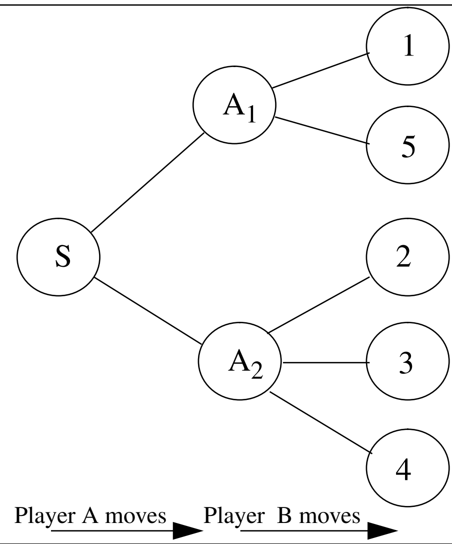
This polymorphic version of the Figure type defines several different types:

- Figure Double — measurements recorded as double-precision, floating point numbers
- Figure Int — measurements recorded as integers
- Figure Rational — measurements recorded as rational numbers

To illustrate the use of defined types in an important area of computing science, consider the problem of analyzing sequences of plays in certain kinds of two-player games. Such games fall into a general pattern that could be called minimax tree-games. Tic-tac-toe, chess, and gin rummy are a few examples. At each stage, one player or the other is obliged to take an action. The rules specify the allowable actions, and each action by one player presents a new stage of the game to the other player. That player is then obliged to select one of the actions permitted by the rules.

The opponents have opposite goals: what is good for one is bad for the other. The software in this lesson will represent these goals as numeric scores. One player will seek to conclude the game with the highest possible score, and the other try to force as low a score as possible.

***Two-move Tree-game***




Game starts in position S.
Player A chooses position $A_1$ or $A_2$.
If Player A chooses $A_1$
then Player B may choose a position with a score of 1 or a position with a score of 5
If Player A chooses $A_2$.
then Player B may has a choice of three positions, one with score 2, another with score 3, and a third with score 4.

**Goals**
Player A — highest possible score
Player B — lowest possible score

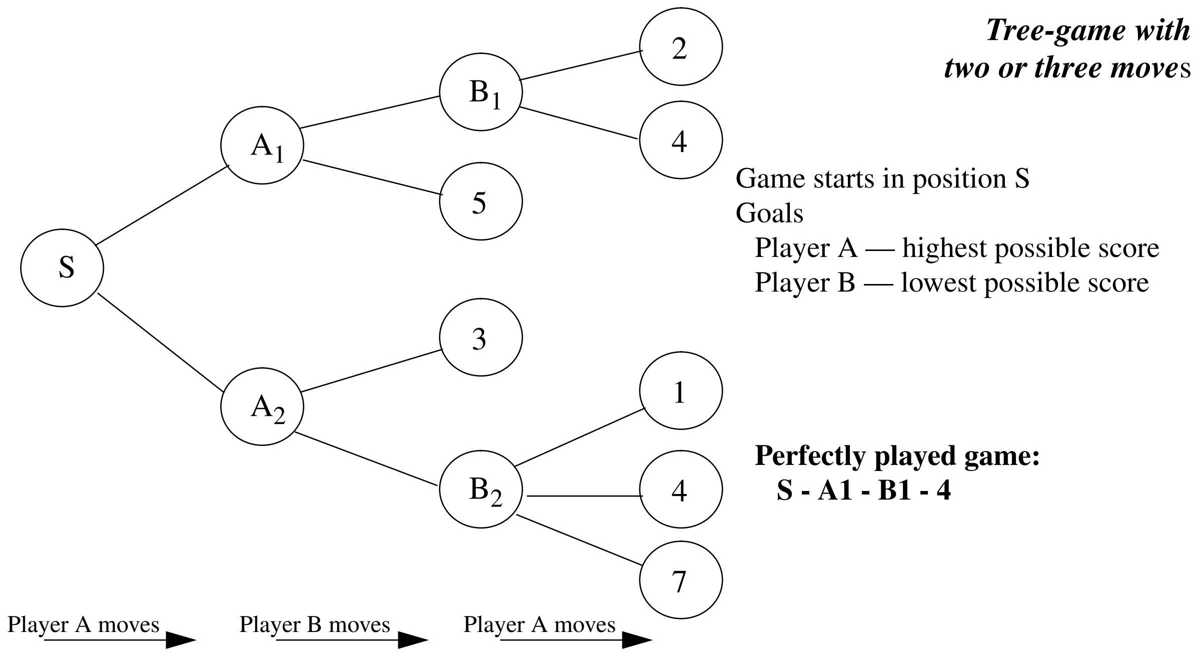
To get a feeling for this model, study the diagram of the two-move tree-game. In this game, Player A, to maximize his score, will choose position $A_2$. From position $A_2$ the worst score Player A can get is 2, while from position $A_1$ he could get a score as low as 1. In fact Player A will definitely get a score of 1 of he moves to position $A_1$ unless Player B makes a mistake.

When Player A chooses position $A_2$, he is using what is known as a **minimax strategy**. He chooses the position that maximizes, over the range of options available, the smallest possible score he could get. Player B uses the same strategy, but inverted. She chooses the position that minimizes, over her range of options, the largest possible score that she might obtain (because her goal is to force the game to end with the lowest score possible).

These games are artificial ones, described directly in terms of diagrams showing possible moves and eventual scores, but the same sort of structure can be used to describe many two-player

***Test your understanding of minimax principles by analyzing this tree game.***

***It requires either two or three moves, depending on which move is chosen first.***




games. If you have following three pieces of information about a game, you can draw diagram for the game similar to these tree-game charts:

1. **moves** — a rule that specifies what moves can take place from a given position,
2 **score** — a function that can compute the score from a position that ends a game, and
3 **player** — a rule that, given a game position, can determine which player is to play next.

Diagrams of this form occur frequently in computer science. They are called tree diagrams, or, more commonly, just **trees**. In general, a tree consists of an entity known as its root, plus a collection of subtrees. A subtree is, itself, a tree.

In these terms, the two-move game in the diagram is a tree with root S and two subtrees. One of the subtrees is a tree with root A1 and two subtrees (each of which has a root and an empty collection of subtrees). The other subtree is a tree with root A2 and three subtrees (each of which, again, has a root and an empty collection of subtrees).

The goal of this chapter will be to write a function that, given the three necessary pieces of information (in the form of other functions: `moves`, `score`, and `player`) and a starting position for a game will build a representation of a tree-diagram, use it to carry out a game played perfectly by both players, and report the position at the end of the game.

One piece of information the software will need to deal with from time to time is the identity of the player whose turn it is to proceed. This information could be represented in terms of intrinsic types in many ways. A player's identity could be known by a character for example, perhaps `'A'` for Player A and `'B'` for Player B. Or, integers could be chosen to designate the players, perhaps `1` for Player A and `2` for Player B.

Instead of using one of these alternatives, the identity of the player will be represented by a newly defined data type called Player. This will take advantage of the Haskell system's type checking facility to keep from mixing up a player's identity with a character or number used for some other purpose. The functions that need the player's identity will get a value of the newly defined type and will not be able to use it as if it were a character or integer or some other type of value. This reduces the number of ways that the program can be in error.

This definition establishes the Player type with two constructors, PlayerA and PlayerB:

HASKELL DEFINITION • `data Player = PlayerA | PlayerB`

A type need not have more than one constructor. For example, the following type will be used to represent game trees.

HASKELL DEFINITION • `data Game position = Plays position [Game position]`

The type Game is polymorphic. The parameter that makes it polymorphic (denoted by the name position in the definition), can be any type. Therefore, Game is really a family types, one for each possible type that position might be (Int, String, [Int], or whatever).

Any value of type Game will be built by the constructor Plays and will take the form of the constructor name Plays followed by a value of type position, followed in turn by a sequence of values of type Game. The definition is recursive, as you might expect it to be, since a game is a tree and a tree is a root and a collection of subtrees.

The name position in the definition of Game is simply a placeholder. A variable of type Game will actually have type Game Int if the placeholder is the type Int. On the other hand, the variable will have type Game [Int] if the placeholder is the type [Int]. The polymorphic nature of the type Game is necessary because the function to be written is supposed to work regardless of the details of the game itself. Different games, of course, would need to record different information to represent a position in the game. One representation of position would not fit all games.

The function to carry out a game from a given position, a function called perfectGameFromPosition, will be packaged in a module called Minimax. Since all computations requiring an understanding of the details of a value of type position will be performed by functions supplied as arguments to perfectGameFromPosition, the module Minimax can treat position in an entirely abstract way. It matters not at all to functions in the module Minimax how the type position is represented.

There are two components of the computation that perfectGameFromPosition carries out: one to generate the game tree and the other to use the minimax strategy to find the final position of a game played perfectly from the point of view of both players.

Consider first the problem of building the game tree. This can be done in stages. Starting from a given position, compute all of the positions attainable in one move from that position. (One of the functions supplied as an argument to perfectGameFromPosition is responsible for delivering this collection of positions — this function is referred to as moves in the module Minimax.)

The positions computed from the initial position become the starting positions of the subtrees of the root in the game tree Their game trees can, of course, be computed in the same way. The computation is recursive in the same way that the type representing game trees is recursive.

```
HASKELL DEFINITION •   gameTree:: (position -> [position]) -> position -> Game position
HASKELL DEFINITION •   gameTree moves p = Plays p (map (gameTree moves) (moves p))
```

Depending on the game, this tree could be infinite, in which case the minimax strategy won't work. To use the minimax strategy, potentially infinite games, such as checkers, must be arbitrarily cut off at some stage by the moves function. (This is what people do, in a sense, when they try to plan ahead a few moves in games like checkers. They analyze the situation as far ahead as they can manage, then guess that the final score will be related to the quality of their position at that point.) However, the game tree will be finite if every route down through the subtrees eventually comes to a tree containing an empty sequence of subtrees.

Now consider the problem of choosing a move from a collection of alternatives in the game tree. If it is Player A's turn to move, he will need to look at the scores Player B could get by making her best move from each of the positions Player A can move to. Once this is computed, all Player A has to do is choose the move that maximizes his score. The following definition of the function play follows this strategy, but only for the case when it is Player A's turn to play. (The function score in this definition is the function supplied to perfectGameFromPosition, which can compute the score in the game, given a game-ending position.)

```
HASKELL DEFINITION •   play PlayerA score (Plays p gs)
HASKELL DEFINITION •     | null gs    = p
HASKELL DEFINITION •     | otherwise  = foldr1 (maxPosition score)
HASKELL DEFINITION •                                        (map (play PlayerB score) gs)
HASKELL DEFINITION •   maxPosition score p q
HASKELL DEFINITION •     | score p > score q   = p
HASKELL DEFINITION •     | otherwise           = q
```

HASKELL DEFINITION • Player B would, of course, follow the same strategy, but looking for a minimal rather than a maximal score:

```
HASKELL DEFINITION •   play PlayerB score (Plays p gs)
HASKELL DEFINITION •     | null gs    = p
HASKELL DEFINITION •     | otherwise  = foldr1 (minPosition score)
HASKELL DEFINITION •                                        (map (play PlayerA score) gs)
HASKELL DEFINITION •   minPosition score p q
HASKELL DEFINITION •     | score p < score q   = p
HASKELL DEFINITION •     | otherwise           = q
```

All that is left to do to put together the function perfectGameFromPosition is to apply the play function to the game tree generated from the initial position supplied as an argument. Try to fill in the definition of perfectGameFromPosition yourself, as part of the module Minimax, which pulls together the functions defined so far in this chapter.

```
¿ HASKELL DEFINITION ?   module Minimax
¿ HASKELL DEFINITION ?     (Player(PlayerA, PlayerB),
¿ HASKELL DEFINITION ?       perfectGameFromPosition)
¿ HASKELL DEFINITION ?     where
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?     data Player = PlayerA | PlayerB
```

```
¿ HASKELL DEFINITION ?    data Game position = Plays position [Game position]
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    perfectGameFromPosition :: Real num =>
¿ HASKELL DEFINITION ?      (position->[position]) -> (position->num) -> (position->Player)
¿ HASKELL DEFINITION ?      -> position -> position
¿ HASKELL DEFINITION ?    perfectGameFromPosition moves score player p =
¿ HASKELL DEFINITION ?                                            --you define this function
¿ HASKELL DEFINITION ?      play (player p) score (gameTree moves p)
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    gameTree:: (position -> [position]) -> position -> Game position
¿ HASKELL DEFINITION ?    gameTree moves p =
¿ HASKELL DEFINITION ?      Plays p (map (gameTree moves) (moves p))
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    play :: Real num =>
¿ HASKELL DEFINITION ?      Player -> (position -> num) -> Game position -> position
¿ HASKELL DEFINITION ?    play PlayerA score (Plays p gs)
¿ HASKELL DEFINITION ?      | null gs    = p
¿ HASKELL DEFINITION ?      | otherwise  = foldr1 (maxPosition score)
¿ HASKELL DEFINITION ?                                         (map (play PlayerB score) gs)
¿ HASKELL DEFINITION ?    play PlayerB score (Plays p gs)
¿ HASKELL DEFINITION ?      | null gs    = p
¿ HASKELL DEFINITION ?      | otherwise  = foldr1 (minPosition score)
¿ HASKELL DEFINITION ?                                         (map (play PlayerA score) gs)
¿ HASKELL DEFINITION ?
¿ HASKELL DEFINITION ?    minPosition, maxPosition:: Real num =>
¿ HASKELL DEFINITION ?      (position -> num) -> position -> position -> position
¿ HASKELL DEFINITION ?    minPosition score p q
¿ HASKELL DEFINITION ?      | score p < score q   = p
¿ HASKELL DEFINITION ?      | otherwise           = q
¿ HASKELL DEFINITION ?    maxPosition score p q
¿ HASKELL DEFINITION ?      | score p > score q   = p
¿ HASKELL DEFINITION ?      | otherwise           = q                                          1
```

A notable feature of the module Minimax is that it exports not only the function that carries out the minimax strategy, but also the type Player and its constructors. This is necessary because any other module using the facilities of Minimax will have to define a function that delivers the identity of the player whose turn it is to play, given a particular position in the game. To supply this function, the module will require access to the type used in module Minimax to represent players.

The other type defined in the module Minimax, that is the type Game position, does not need to be visible outside the module. So, Game position is not exported. The module Minimax does not import the facilities of any other module, but it will inherit the type position from any module that uses Minimax to carry out a game computation. In this sense, the type position, is abstract with respect to the module Minimax, while the type Player is concrete in Minimax and will also be concrete in any module using Minimax.

To see how the module Minimax can be used, consider the game of tic-tac-toe. Players take turns marking squares on a three-by-three grid. If one player marks three squares in a line (horizontally, vertically, or diagonally), that player wins. The game is sometimes called noughts and crosses because the first player to mark the grid normally marks with an X, the other an O.

One way to represent a position in tic-tac-toe is to use a sequence of nine integers. The first three positions in the sequence represent the top row of the grid, the next three the middle row, and the last three the bottom row. If an integer in the sequence is zero, it indicates that the corresponding square in the grid is unmarked. If the integer is a non-zero value $n$, it indicates that the corresponding square was marked in the $n^{th}$ move of the game.

> ***position and game history***
> The minimax computation delivers the final position of a game played perfectly from a supplied starting position. Normally, one would like to see the sequence of moves leading to the final position. One way to get that information is to design the representation of positions so that each position contains the entire sequence of moves leading up to it. The encoding chosen for TicTacToePosition follows this strategy.

From this representation, you can figure out which player marked each square: if the integer is odd, the X player marked it, and if it is even the O player marked it. You can also figure out which player's turn it is to play (the largest integer in the grid indicates which player played last — the other player is next to play). This provides a way to write the necessary player function:

```
HASKELL DEFINITION •   ticTacToePlayer(Grid g)
HASKELL DEFINITION •     | even(maximum g)  = PlayerA
HASKELL DEFINITION •     | otherwise        = PlayerB
```

You can also determine from a position represented in this form whether or not the game is over and, if it is over, which player won. To do this, just extract from the grid each of the triples of integers corresponding to eight straight lines through the grid (top row, middle row, bottom row, left column, middle column, right column, diagonal, and back diagonal).[1] Then check to see of any of these triples contains three X's (odd integers) or three O's (even integers other than zero).

> **odd :: Integral num => num -> Bool**
> **even :: Integral num => num -> Bool**
> *intrinsic functions*
> odd = True *iff argument is an odd integer*
> even = not . odd

If there are three X's in a row, then X wins; score that as 1. If there are three O's in a row, then O wins; score that a -1 (since the Minimax module is set up so that PlayerB, the name it uses for the O player, tries to force the game to a minimum score). If the grid is entirely marked with X's and O's and there is no place left to mark, then the game is over, and it is a draw; score that as zero.

1. These elements of the grid could be extracted using combinations of head and tail, but it is more concise to use the indexing operator (!!). If xs is a sequence and n is an integer, the xs!!n is element n of xs. Elements are numbered starting from zero, so xs!!0 is head(xs), xs!!1 is head(tail(xs)), and so on. Of course, xs!!n is not defined if xs has no element n.

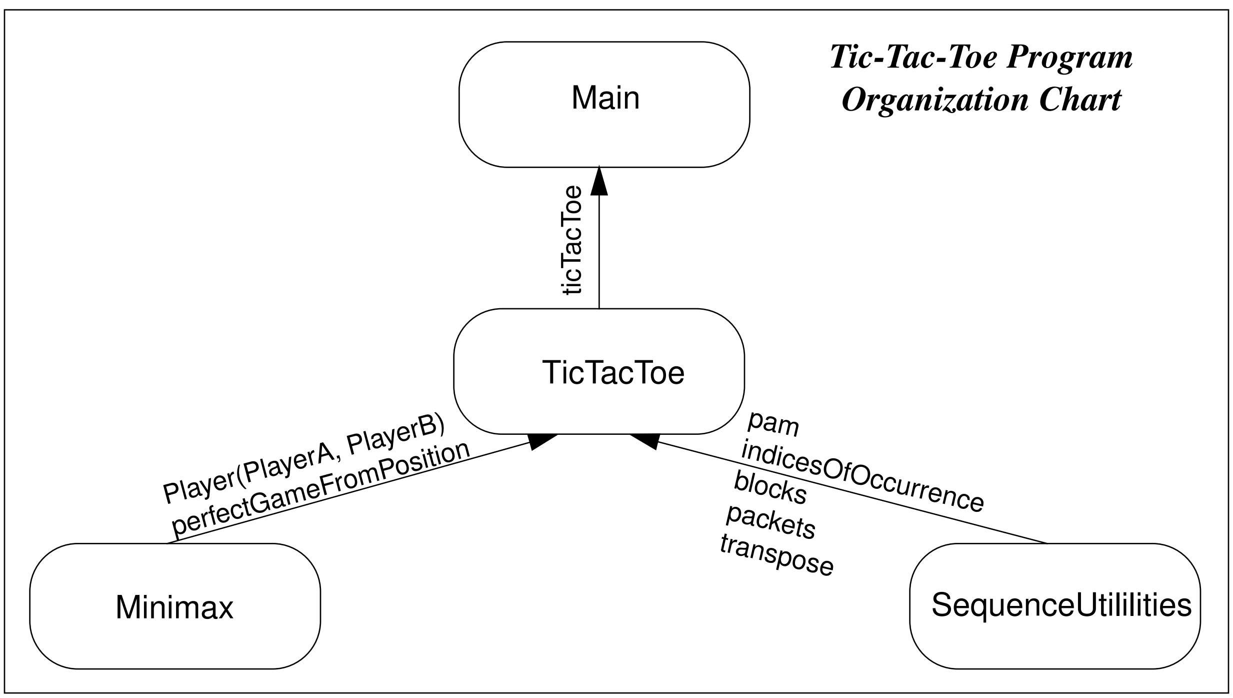
HASKELL DEFINITION • ticTacToeScore p
HASKELL DEFINITION • | win PlayerA p = 1
HASKELL DEFINITION • | win PlayerB p = -1
HASKELL DEFINITION • | otherwise = 0

The win function used in the definition of ticTacToeScore is a bit awkward because it has to extract all the lines from the grid and deal with other technicalities. Nevertheless, it follows the above outline in a straightforward way. You can work out the details for yourself more easily than you can read an explanation of them.

The other function that the Minimax module uses to carry out the minimax calculation is the function that generates the possible moves for a player from a given position. Since a player can make a mark in any open square, this computation amounts to locating the unmarked squares, that is the squares with zeros in them. Given an existing position and the location of an open square, you can build a new position by copying the grid representing the old one, except that in the open square, you put an integer that is one greater than the largest integer in the existing grid.

HASKELL DEFINITION • ticTacToeMoves:: TicTacToePosition -> [TicTacToePosition]
HASKELL DEFINITION • ticTacToeMoves p
HASKELL DEFINITION • | ticTacToeGameOver p = [ ]
HASKELL DEFINITION • | otherwise = map (makeMark p) (openSquares p)
HASKELL DEFINITION •

Again, the details (buried in the functions makeMark and openSquares) are more easily understood by working them out for yourself than by reading someone else's explanation..




The preceding explanation will help you work your way through the following module. It imports several functions from the SequenceUtilities module (in the Appendix). And, you will need to either work out for yourself some way to display the information in a grid, or just accept the showGrid function defined in the module as a suitable display generator. It builds a three-line sequence containing a picture of the grid marked with X's and O's and another picture marked with integers, so you can follow the progress of the game.

```
HASKELL DEFINITION •   module TicTacToe(ticTacToe)
HASKELL DEFINITION •     where
HASKELL DEFINITION •     import Minimax
HASKELL DEFINITION •       (Player(PlayerA, PlayerB), perfectGameFromPosition)
HASKELL DEFINITION •     import SequenceUtilities
HASKELL DEFINITION •       (pam, indicesOfOccurence, blocks, packets, transpose)
HASKELL DEFINITION •     import Char(toUpper)
HASKELL DEFINITION •
HASKELL DEFINITION •     ticTacToe =
HASKELL DEFINITION •       showGrid .
HASKELL DEFINITION •       perfectGameFromPosition
HASKELL DEFINITION •                ticTacToeMoves ticTacToeScore ticTacToePlayer .
HASKELL DEFINITION •       positionFromString
HASKELL DEFINITION •
HASKELL DEFINITION •     data TicTacToePosition  = Grid [Int]
HASKELL DEFINITION •                                  -- Grid g :: TicTacToePostition means
HASKELL DEFINITION •                                 -- g = [mark-1, mark-2, ..., mark-9] and
HASKELL DEFINITION •                                                          -- 0 <= mark-i <= 9
HASKELL DEFINITION •                                        -- mark-i = 0 means empty square
HASKELL DEFINITION •                                 -- mark-i = odd means X occupies square
HASKELL DEFINITION •                             -- mark-i = even, > 0 means O occupies square
HASKELL DEFINITION •
HASKELL DEFINITION •     data Gridline = Slice [Int]
HASKELL DEFINITION •                               -- row, column, or diagonal of grid (length 3)
HASKELL DEFINITION •                           -- Slice [mark-1, mark-2, mark-3] :: Gridline means
HASKELL DEFINITION •                                                          -- 0 <= mark-i <= 9
HASKELL DEFINITION •
HASKELL DEFINITION •     positionFromString :: String -> TicTacToePosition
HASKELL DEFINITION •     positionFromString =
HASKELL DEFINITION •       Grid . map intFromDigit . takeWhile(/= '.') .
HASKELL DEFINITION •       convert '#' empties .
HASKELL DEFINITION •       convert 'O' movesO .
HASKELL DEFINITION •       convert 'X' movesX .
HASKELL DEFINITION •       (++ ".") . filter(`elem` "XO#") . map toUpper
HASKELL DEFINITION •       where
HASKELL DEFINITION •       empties = repeat '0'
HASKELL DEFINITION •       movesX = "13579"
HASKELL DEFINITION •       movesO = "2468"
HASKELL DEFINITION •
HASKELL DEFINITION •     intFromDigit :: Char -> Int
HASKELL DEFINITION •     intFromDigit digit = fromEnum(digit) - fromEnum('0')
HASKELL DEFINITION •
HASKELL DEFINITION •     convert :: Char -> String -> String -> String
```

```
HASKELL DEFINITION •    convert thisMoveSymbol moveDigits =
HASKELL DEFINITION •      concat . zipWith pasteIn moveDigits .
HASKELL DEFINITION •                                  packets(== thisMoveSymbol)
HASKELL DEFINITION •      where
HASKELL DEFINITION •      pasteIn moveDigit otherMoveSymbols =
HASKELL DEFINITION •                              otherMoveSymbols ++ [moveDigit]
HASKELL DEFINITION •
HASKELL DEFINITION •    ticTacToeMoves:: TicTacToePosition -> [TicTacToePosition]
HASKELL DEFINITION •    ticTacToeMoves p
HASKELL DEFINITION •      | ticTacToeGameOver p      = [ ]
HASKELL DEFINITION •      | otherwise                = map (makeMark p) (openSquares p)
HASKELL DEFINITION •
HASKELL DEFINITION •    ticTacToeScore:: TicTacToePosition -> Int
HASKELL DEFINITION •    ticTacToeScore p
HASKELL DEFINITION •      | win PlayerA p    =  1
HASKELL DEFINITION •      | win PlayerB p    = -1
HASKELL DEFINITION •      | otherwise        =  0
HASKELL DEFINITION •
HASKELL DEFINITION •    ticTacToeGameOver:: TicTacToePosition -> Bool
HASKELL DEFINITION •    ticTacToeGameOver =
HASKELL DEFINITION •      or . pam[gridFull, win PlayerA, win PlayerB]
HASKELL DEFINITION •
HASKELL DEFINITION •    openSquares:: TicTacToePosition -> [Int]
HASKELL DEFINITION •    openSquares(Grid g) = indicesOfOccurence 0 g
HASKELL DEFINITION •
HASKELL DEFINITION •    makeMark:: TicTacToePosition -> Int -> TicTacToePosition
HASKELL DEFINITION •    makeMark (Grid g) indexOfSquare =
HASKELL DEFINITION •      Grid(take indexOfSquare g ++ [maximum g + 1] ++
HASKELL DEFINITION •      drop (indexOfSquare + 1) g)
HASKELL DEFINITION •
HASKELL DEFINITION •    diag, backdiag, toprow, midrow, botrow, lftcol, midcol, rgtcol ::
HASKELL DEFINITION •      TicTacToePosition -> Gridline
HASKELL DEFINITION •    diag(Grid g)                        = Slice[g!!0, g!!4, g!!8]
HASKELL DEFINITION •    backdiag(Grid g)                    = Slice[g!!2, g!!4, g!!6]
HASKELL DEFINITION •    toprow(Grid g)                      = Slice[g!!0, g!!1, g!!2]
HASKELL DEFINITION •    midrow(Grid g)                      = Slice[g!!3, g!!4, g!!5]
HASKELL DEFINITION •    botrow(Grid g)                      = Slice[g!!6, g!!7, g!!8]
HASKELL DEFINITION •    lftcol(Grid g)                      = Slice[g!!0, g!!3, g!!6]
HASKELL DEFINITION •    midcol(Grid g)                      = Slice[g!!1, g!!4, g!!7]
HASKELL DEFINITION •    rgtcol(Grid g)                      = Slice[g!!2, g!!5, g!!8]
HASKELL DEFINITION •
HASKELL DEFINITION •    gridlines:: [TicTacToePosition -> Gridline]
HASKELL DEFINITION •    gridlines = [diag, backdiag,
HASKELL DEFINITION •                     toprow, midrow, botrow,  lftcol, midcol, rgtcol]
HASKELL DEFINITION •
HASKELL DEFINITION •    gridlineFilledByPlayer :: Player -> Gridline -> Bool
```

```
HASKELL DEFINITION •   gridlineFilledByPlayer PlayerA (Slice s) = (and . map odd) s
HASKELL DEFINITION •   gridlineFilledByPlayer PlayerB (Slice s) =
HASKELL DEFINITION •     (and . map positiveEven) s
HASKELL DEFINITION •     where
HASKELL DEFINITION •     positiveEven k = k > 0  &&  even k
HASKELL DEFINITION •
HASKELL DEFINITION •   win:: Player -> TicTacToePosition -> Bool
HASKELL DEFINITION •   win player =
HASKELL DEFINITION •     or . map(gridlineFilledByPlayer player) . pam gridlines
HASKELL DEFINITION •
HASKELL DEFINITION •   gridFull:: TicTacToePosition -> Bool
HASKELL DEFINITION •   gridFull(Grid g)   =   maximum g == 9
HASKELL DEFINITION •
HASKELL DEFINITION •   showGrid:: TicTacToePosition -> String
HASKELL DEFINITION •   showGrid(Grid g) =
HASKELL DEFINITION •     (unlines . map concat . transpose)
HASKELL DEFINITION •        [gridMarkedXO,  map ("  "++) gridMarkedByMoveNumber]
HASKELL DEFINITION •     where
HASKELL DEFINITION •     gridMarkedXO  = blocks 3 (map markFromMoveNumber g)
HASKELL DEFINITION •     gridMarkedByMoveNumber =
HASKELL DEFINITION •                        blocks 3 (map digitFromMoveNumber g)
HASKELL DEFINITION •     markFromMoveNumber m
HASKELL DEFINITION •                                             | m == 0     = '#'
HASKELL DEFINITION •                                             | odd m      = 'X'
HASKELL DEFINITION •                                             | otherwise  = 'O'
HASKELL DEFINITION •     digitFromMoveNumber m
HASKELL DEFINITION •                                             | m == 0     = '#'
HASKELL DEFINITION •                                  | otherwise  = head(show m)
HASKELL DEFINITION •
HASKELL DEFINITION •   ticTacToePlayer :: TicTacToePosition -> Player
HASKELL DEFINITION •   ticTacToePlayer(Grid g)
HASKELL DEFINITION •     | odd(maximum g)  = PlayerB
HASKELL DEFINITION •     | otherwise       = PlayerA                                2
```

The following module imports the tic-tac-toe module and defines a few game setups.The commands then show the results that the minimax strategy produces for these situations. The first two of the setups begin from a partially played game, played imperfectly, to show that the minimax strategy will if it has an opportunity.

```
HASKELL DEFINITION •  import TicTacToe(ticTacToe)
HASKELL DEFINITION •
HASKELL DEFINITION •  advantageO =
HASKELL DEFINITION •    "XX#" ++
HASKELL DEFINITION •    "###" ++
HASKELL DEFINITION •    "##O"
HASKELL DEFINITION •
HASKELL DEFINITION •  advantageX =
```

```
HASKELL DEFINITION •      "X##" ++
HASKELL DEFINITION •      "###" ++
HASKELL DEFINITION •      "O##"
HASKELL DEFINITION •
HASKELL DEFINITION •   cat8 =
HASKELL DEFINITION •      "###" ++
HASKELL DEFINITION •      "#X#" ++
HASKELL DEFINITION •      "###"
HASKELL DEFINITION •
HASKELL DEFINITION •   cat9 =
HASKELL DEFINITION •      "###" ++
HASKELL DEFINITION •      "###" ++
HASKELL DEFINITION •      "###"
HASKELL DEFINITION •
```

2a

```
HASKELL COMMAND •   putStr(ticTacToe advantageO)
HASKELL RESPONSE •  XXO  134
HASKELL RESPONSE •  ##O  ##6
HASKELL RESPONSE •  #XO  #52
HASKELL COMMAND •   putStr(ticTacToe advantageX)
HASKELL RESPONSE •  X#X  1#7
HASKELL RESPONSE •  #OX  #65
HASKELL RESPONSE •  OOX  243
HASKELL COMMAND •   putStr(ticTacToe cat8)
HASKELL RESPONSE •  XOO  948
HASKELL RESPONSE •  OXX  615
HASKELL RESPONSE •  XXO  732
HASKELL COMMAND •   putStr(ticTacToe cat9)
HASKELL RESPONSE •  XOX  985
HASKELL RESPONSE •  XOO  726
HASKELL RESPONSE •  OXX  431
```

Some of the game sequences generated by minimax analysis may look like one player is intentionally throwing the game. When there are more routes than one to a win for one player or the other, the minimax computation will select one of those routes, without regard to whether it may or may not look competitive to an experienced player. The essential fact is this: when one player is in a position to win, there is nothing the other player can do to keep that player from winning. So, the losing player can make arbitrary moves without affecting the result. The minimax strategy examines all of the relevant possibilities, but the game it selects as its route to the end could be any of the possible routes. The winning player will never give the losing player an opportunity to win. But, the player in a losing position may give the other player an opportunity to win easily.

Finally, you may be interested in knowing that most game playing programs, such as chess players, checkers players, go players, backgammon players, and so on, use the minimax strategy for at least part of their analysis. However, they use a form of the computation that involves substantially less computation.

This more efficient form of the computation is known as the alpha-beta algorithm. It looks ahead in the game tree and eliminates, without further analysis on the subtree, options that cannot improve the situation for a given player.[1] This almost always makes it possible to complete the computation in something like a small multiple of the square root of the time it would take using the naive form of the minimax algorithm (the one presented in this chapter). The analysis can then proceed about twice as far down the game tree as it could have with naive minimax analysis.

Nevertheless, even with the alpha-beta form of minimax analysis, it is impractical to analyze very deeply in game trees for large games like chess because such game trees increase in size so rapidly with the number of moves analyzed that even the square root of the minimax time is still impossibly long. So, practical game playing programs combine minimax analysis (in its alpha-beta form) with specialized analysis methods designed around particular approaches to playing the game.

## *Review Questions*

1 A tree, in computer science, is an entity
- a with a root and two subtrees
- b with a root and a collection of subtrees, each of which is also a tree ← *correct answer*
- c with a collection of subtrees, each of which has one or more roots
- d described in a diagram with circles, lines, and random connections

2 A sequence, in Haskell, is an entity
- a with one or more elements
- b that is empty or has a first element followed by a sequence of elements ← *correct answer*
- c whose elements are also sequences
- d with a head and one or more tails

3 The following definition specifies

```
HASKELL DEFINITION •    data WeekDay =
HASKELL DEFINITION •        Monday | Tuesday | Wednesday | Thursday | Friday
```

- a a type with five constructors ← *correct answer*
- b a type with five explicit constructors and two implicit ones
- c a tree with five roots
- d a sequence with five elements

4 Given the definition in the preceding question, what is the type of the following function `f`?

```
HASKELL DEFINITION •    f Tuesday = "Belgium"
```

- a `f :: WeekDay -> String` ← *correct answer*
- b `f :: Tuesday -> "Belgium"`
- c `f :: Day -> Country`
- d type of `f` cannot be determined

1. You can find out how it does this in any standard text on artificial intelligence. Also, the text *Introduction to Functional Programming* by Bird and Wadler, Prentice-Hall, 1988, contains an elegant derivation of the alpha-beta algorithm as a Haskell-like program from a form of the minimax program similar to the one in this chapter.

5 Types defined in Haskell scripts with the data keyword
  a must begin with a capital letter
  b may be imported from modules
  c must be used consistently in formulas, just like intrinsic types
  d all of the above ← *correct answer*

6 What kind of structure does the following type represent?
  *HASKELL DEFINITION •* data BinaryTree = Branch BinaryTree BinaryTree | Leaf String
  a a type with four constructors
  b a digital structure
  c a tree made up of ones and zeros
  d a tree in which each root has either two subtrees or none ← *correct answer*

7 Given the preceding definition of the type BinaryTree, which of the following defines a function that computes the total number of Branch constructors in an entity of type BinaryTree?
  a branches binaryTree = 2
  b branches (Branch left right) = 2
  branches (Leaf x) = 0
  c branches (Branch left right) = 1 + branches left + branches right ← *correct answer*
  branches (Leaf x) = 0
  d branches (Branch left right) = 2∗branches left + 2∗branches right
  branches (Leaf x) = 1

8 The formula xs!!(length xs - 1)
  a is recursive
  b has the same type as xs
  c delivers the same result as last xs ← *correct answer*
  d none of the above

9 Given the definition of the function pam in the module SequenceUtilities, the formula
  pam (map (+) [1 . . 5]) 10
  a delivers the same result as map (1+) [1 . . 5]
  b delivers the same result as pam [1 .. 5] (map (1+))
  c delivers the result [11, 12, 13, 14, 15] ← *correct answer*
  d all of the above

10 Given the Grid [1,3,0, 0,0,0, 0,0,2] (as in the tic-tac-toe script), what is the status of the game?
  a game over, X wins
  b game over, O wins
  c O's turn to play ← *correct answer*
  d X's turn to play

11 Which of the following formulas extracts the diagonal of a grid (as in the tic-tac-toe program)?
  a (take 3 . map head . iterate(drop 4)) grid
  b [head grid, head(drop 4 grid), head(drop 8 grid)] ← *correct answer*
  c [head grid, grid!!4, last(grid)]
  d all of the above