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 ? 7

HASKELL COMMAND • reverse notPalindrome 8

¿ HASKELL RESPONSE ? 9

HASKELL COMMAND • longPalindrome == reverse longPalindrome 10

¿ HASKELL RESPONSE ? 11

HASKELL COMMAND • notPalindrome == reverse notPalindrome 12

¿ HASKELL RESPONSE ? 13

HASKELL COMMAND • longPalindrome == shortPalindrome 14

¿ HASKELL RESPONSE ? 15

HASKELL COMMAND • reverse squashedPalindrome == squashedPalindrome 16

¿ HASKELL RESPONSE ? 17

HASKELL COMMAND • "ABLE WAS I ERE I SAW ELBA" == spacedPalindrome 18

¿ HASKELL RESPONSE ? 19

¿ HASKELL DEFINITION ? 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 ? 32

HASKELL COMMAND • isPalindrome notPalindrome 33

¿ HASKELL RESPONSE ? 34

HASKELL COMMAND • isPalindrome squashedPalindrome 35

¿ HASKELL RESPONSE ? 36

HASKELL COMMAND • isPalindrome (reverse shortPalindrome) 37

¿ HASKELL RESPONSE ? 38

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

By this rule,

```
isPalindrome reverse shortPalindrome
```

would mean

```
(isPalindrome reverse) shortPalindrome
```

rather than

```
isPalindrome (reverse shortPalindrome).
```

The parentheses are necessary to get the intended meaning.

> ***Haskell programs = collections of definitions***
>
> Haskell programs are collections of definitions. When you construct software in Haskell, you will be defining the meaning of a collection of terms. Most of these terms will be functions, and you will define these functions as parameterized formulas that say what value the function should deliver.
>
> People using a Haskell program write Haskell commands specifying the computation they want the computer to perform. These commands are formulas written in terms of the functions defined in a program.
>
> Definitions, therefore, form the basis for all software construction in Haskell.

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

HASKELL DEFINITION • `word = reverse "drow"`

HASKELL COMMAND • `word`

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

2 How about this command?

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

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

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

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

HASKELL COMMAND • `isTrue word`

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

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 ? 5

HASKELL COMMAND • removeBlanks "s p a c e d o u t" 6

¿ HASKELL RESPONSE ? 7

HASKELL COMMAND • removeBlanks maddog 8

¿ HASKELL RESPONSE ? 9

HASKELL COMMAND • removeBlanks hot 10

¿ HASKELL RESPONSE ? 11

HASKELL COMMAND • removeBlanks(reverse chicago) 12

¿ HASKELL RESPONSE ? 13

HASKELL COMMAND • removeBlanks hot == reverse(removeBlanks hot) 14

¿ HASKELL RESPONSE ? 15

¿ HASKELL DEFINITION ? -- function to remove periods (you write it)

¿ HASKELL DEFINITION ? removePeriods str =

¿ HASKELL DEFINITION ? 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 ? 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
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

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 ?                                                          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 ?                                                              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.

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

HASKELL DEFINITION • `f  str  =  [ c | c <- str, c == 'x' ]`

HASKELL DEFINITION • `g  str  =  [ c | c <- reverse str, c < 'n' ]`

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

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

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

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

HASKELL COMMAND • `(f . g)  teddy`

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

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

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

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

4 What would be the answer to the preceding question if, in the definitions of f and g, the sense of all the comparisons had been reversed (not equals instead of equals, less-than instead of greater-than-or-equal, etc.)?

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

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

a no arguments
b one argument
c two arguments
d three arguments

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 ? 6

HASKELL COMMAND • :type [True, False, False] 7

¿ HASKELL RESPONSE ? 8

HASKELL COMMAND • :type ["aardvark", "boar", "cheeta"] 9

¿ HASKELL RESPONSE ? 10

HASKELL COMMAND • reverse ["aardvark", "boar", "cheeta"] 13

¿ HASKELL RESPONSE ? 14

HASKELL COMMAND • :type reverse ["aardvark", "boar", "cheeta"] 11

¿ HASKELL RESPONSE ? 12

Review Questions

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

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

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

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

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

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

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

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 ? 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.

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
- d gradually change shape as the computation proceeds

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

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

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

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

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

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

8 A function with the type `Eq a => a -> Bool`

- a requires an argument with the name `a`
- b delivers `True` on arguments of type `a`
- c is polymorphic
- d must be equal to `a`

9 If the type of `f` has three arrows in it, then the type of `f x` has

- a one arrow in it
- b two arrows in it
- c three arrows in it
- d four arrows in it

10 A polymorphic function

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

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 ?
HASKELL COMMAND •   [Red .. Blue]
¿ HASKELL RESPONSE ?
HASKELL COMMAND •   Circle Red 1  ==  Circle Red 2
¿ HASKELL RESPONSE ?
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)
```