The Haskell School of Music

From Signals to Symphonies

Paul Hudak
Donya Quick

The Haskell School of Music

From Signals to Symphonies

PAUL HUDAK

DONYA QUICK
Stevens Institute of Technology

CAMBRIDGE
UNIVERSITY PRESS

Copyrighted material

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

79 Anson Road, #06–04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781108416757
DOI: 10.1017/9781108241861

© Paul Hudak and Donya Quick 2018

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2018

Printed in the United States of America by Sheridan Books, Inc.

A catalogue record for this publication is available from the British Library.

Library of Congress Cataloging-in-Publication Data
Names: Hudak, Paul, author. | Quick, Donya, author.
Title: The Haskell school of music : from signals to symphonies / Paul Hudak, Donya Quick.
Description: Cambridge, United Kingdom ; New York, NY :
Cambridge University Press, 2018. | Includes bibliographical references and index.
Identifiers: LCCN 2018016879 | ISBN 9781108416757 (hardback : alk. paper)
Subjects: LCSH: Haskell (Computer program language) | Computer music–Instruction and study. | Functional programming (Computer science)
Classification: LCC ML74.4.H37 H84 2018 | DDC 781.3/45133–dc23
LC record available at https://lccn.loc.gov/2018016879

ISBN 978-1-108-41675-7 Hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Copyrighted material

Contents

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

Copyrighted material

musical experience (such as having taken a music appreciation course in school or played an instrument at some point) who want to then explore music in the context of a functional programming environment. Examples of musical concepts that are considered prerequisites to this text are reading Western music notation, the naming scheme for musical pitches as letters and octave numbers, and the major and minor scales. That said, it is certainly not impossible to learn Haskell and the Euterpea library from this book as a complete musical novice – but you will likely need to consult other music-related resources to fill in the gaps as you go along using a dictionary of musical terms. A wide array of free music theory resources and tutorials for beginners are also freely available online. Links to some useful music references and tutorials can be found on the Euterpea website, www.euterpea.com.

Music Terminology

Some musical concepts have more than one term to refer to them, and which synonym is preferred differs by region. For example, the following terms are synonyms for note durations:

American English	**British English**
Double whole note	Breve
Whole note	Semibreve
Half note	Minim
Quarter note	Crotchet
Eight note	Quaver
Sixteenth note	Semiquaver

This book uses the American English versions of these musical terms. The reason for this is that they more closely mirror the mathematical relationships represented by the concepts they refer to, and they are also the basis for names of a number of values used in the software this text describes. The American English standard for naming note durations is both more common in computer music literature and easier to remember for those with limited musical experience – who may struggle to remember what a hemidemisemiquaver is.

Software

There are several implementations of Haskell, all available free on the Internet through the Haskell website, haskell.org. However, the one that has dominated all others, and on which Euterpea is based, is *GHC* [1], an easy-to-use and easy-to-install Haskell compiler and interpreter. GHC runs on a variety of

Copyrighted material

platforms, including Windows, Linux, and Mac. The preferred way to install GHC is using *Haskell Platform* [2]. Once Haskell is installed, you will have access to what is referred to as the *Standard Prelude*, a collection of predefined definitions that are always available and do not need to be specially imported.

Two libraries are needed to code along with this textbook: Euterpea and HSoM. Euterpea is a language for representing musical structures in Haskell, and many of its features are covered in this book. HSoM is a supplemental library containing many of the longer code examples in the text and two additional features: support for modeling musical performance (Chapter 9) and music-related graphical widgets (Chapter 17).

Detailed setup information for Haskell Platform, Euterpea, and HSoM is available on the Euterpea website: www.euterpea.com. Please note: software setup details for Haskell Platform and the Euterpea library varies by architecture (32-bit vs 64-bit), operating system, and compiler version. As the exact setup details are subject to change with every new release of Euterpea's dependencies, please see www.euterpea.com for the most up-to-date installation instructions. While most installations go smoothly with the relatively simple instructions described in the next section, there are many potential differences from one machine to another that can complicate the process. The Euterpea website also contains troubleshooting information for commonly encountered installation problems.

Installation Instructions

The following setup instructions require an Internet connection.

- Download the appropriate version of Haskell Platform from www.haskell.org/platform/ and install it on your machine.
- Open a command prompt (Windows) or terminal (Mac/Linux) and run the following commands:
  ```
  cabal update
  cabal install Euterpea
  cabal install HSoM
  ```
- Mac and Linux users will also need to install a MIDI software synthesizer. Please see the Euterpea website for instructions on how to do this.

The Euterpea website also contains basic walkthroughs for getting started working with the GHC compiler and interpreter within a command prompt or terminal, loading source code files, and so on.

Copyrighted material

Quick References

Brief references for the more commonly used features of Haskell, Euterpea, and HSoM are listed in Appendices E, F, and G. These are intended to serve as a fast way to look up function and value names when you already know a bit about how to use them. A note to students: these few pages of ultra-condensed material are not a substitute for reading the chapters!

Coding and Debugging

Errors are an inevitable part of coding. The best way to minimize the number of errors you have to solve is to code a little bit and then immediately test what you've done. If it's broken, don't wait – fix it then and there! Never press onward and try to work on other things within a file that is broken elsewhere. The reason for this is that one simple error can end up masking others. When the compiler hits a serious problem, it *may not even look at the rest of your file*. As a result, continuing to code without resolving error messages often results in an explosion of new errors once the original one is fixed. You will save yourself a lot of grief by developing good habits of incremental development and not allowing errors to linger unsolved.

Coding *style* is also important. There are two reasons for this in Haskell. The first is that Haskell is *extremely* sensitive to white space characters. Do not mix spaces and tabs! Pick one and be consistent (spaces are typically recommended). Indentation matters, and a small misalignment can sometimes cause bizarre-looking error messages. Style is important, as is readability, both by other programmers and by yourself at a later date. Many novice programmers neglect good coding hygiene, which involves naming things well, laying out code cleanly, and documenting complicated parts of the code. This extra work may be tedious, but it is worthwhile. Coding large projects is often very much dependent on the immediate state of mind. Without that frame of reference, it's not impossible that you could find your own code to be impenetrable if you pick it up again later.

Copyrighted material

Acknowledgments

I wish to thank my funding agencies – the National Science Foundation, the Defense Advanced Research Projects Agency, and Microsoft Research – for their generous support of research that contributed to the foundations of Euterpea. Yale University has provided me a stimulating and flexible environment to pursue my dreams for more than thirty years, and I am especially thankful for its recent support of the Computing and the Arts initiative.

Tom Makucevich, a talented computer music practitioner and composer in New Haven, was the original motivator, and first user, of Haskore, which preceded Euterpea. Watching him toil endlessly with low-level csound programs was simply too much for me to bear! Several undergraduate students at Yale contributed to the original design and implementation of Haskore. I would like to thank in particular the contributions of Syam Gadde and Bo Whong, who coauthored the original paper on Haskore. Additionally, Matt Zamec helped me greatly in the creation of HasSound.

I wish to thank my more recent graduate students, in particular Hai (Paul) Liu, Eric Cheng, Donya Quick, and Daniel Winograd-Cort, for their help in writing much of the code that constitutes the current Euterpea library. In addition, many students in my computer music classes at Yale provided valuable feedback through earlier drafts of the manuscript.

Finally, I wish to thank my wife, Cathy Van Dyke, my best friend and ardent supporter, whose love, patience, and understanding have helped me get through some bad times, and enjoy the good.

Happy Haskell Music Making!

Paul Hudak,
January 2012

Copyrighted material

1
Computer Music, Euterpea, and Haskell

Many computer scientists and mathematicians have a serious interest in music, and it seems that those with a strong affinity or acuity in one of these disciplines is often strong in the other as well. It is quite natural then to consider how the two might interact. In fact, there is a long history of interactions between music and mathematics, dating back to the Greeks' construction of musical scales based on arithmetic relationships, and including many classical composers use of mathematical structures, the formal harmonic analysis of music, and many modern music composition techniques. Advanced music theory uses ideas from diverse branches of mathematics such as number theory, abstract algebra, topology, category theory, calculus, and so on.

There is also a long history of efforts to combine computers and music. Most consumer electronics today are digital, as are most forms of audio processing and recording. But, in addition, digital musical instruments provide new modes of expression, notation software and sequencers have become standard tools for the working musician, and those with the most computer science savvy use computers to explore new modes of composition, transformation, performance, and analysis.

This textbook explores the fundamentals of computer music using a programming-language-centric approach. In particular, the functional programming language *Haskell* is used to express all of the computer music concepts. Thus, a by-product of learning computer music concepts will be learning how to program in Haskell. The core musical ideas are collected into a Haskell library called *Euterpea*. The name "Euterpea" is derived from *Euterpe*, who was one of the nine Greek muses, or goddesses of the arts, specifically the muse of music.

Copyrighted material

say much about what the composer thought as he or she wrote the music, but a program often does. So, when you write your programs, write them for others to see and aim for elegance and beauty, just like the musical result that you desire.

Programming is itself a creative process. Sometimes programming solutions (or artistic creations) come to mind all at once, with little effort. More often, however, they are discovered only after lots of hard work! We may write a program, modify it, throw it away and start over, give up, start again, and so on. It is important to realize that such hard work and reworking of programs is the norm, and in fact you are encouraged to get into the habit of doing so. Do not always be satisfied with your first solution, and always be prepared to go back and change or even throw away those parts of your program that you are not happy with.

1.3 Computation by Calculation

It is helpful when learning a new programming language to have a good grasp of how programs in that language are executed – in other words, an understanding of what a program *means*. The execution of Haskell programs is perhaps best understood as *computation by calculation*. Programs in Haskell can be viewed as *functions* whose input is that of the problem being solved, and whose output is the desired result – and the behavior of functions can be effectively understood as computation by calculation.

An example involving numbers might help demonstrate these ideas. Numbers are used in many applications, and computer music is no exception. For example, integers might be used to represent pitch, and floating-point numbers might be used to perform calculations involving frequency or amplitude.

Suppose we wish to perform an arithmetic calculation such as $3 \times (9 + 5)$. In Haskell this would be written as $3 * (9 + 5)$, since most standard computer keyboards and text editors do not support the special $\times$ symbol. The result can be calculated as follows:

$$\begin{aligned}&3 * (9 + 5)\\&\Rightarrow 3 * 14\\&\Rightarrow 42\end{aligned}$$

It turns out that this is not the only way to compute the result, as evidenced by this alternative calculation:[2]

[2] This assumes that multiplication distributes over addition in the number system being used, a point that will be returned to later in the text.

Copyrighted material

$$\begin{array}{l} 3 * (9 + 5) \\ \Rightarrow 3 * 9 + 3 * 5 \\ \Rightarrow 27 + 3 * 5 \\ \Rightarrow 27 + 15 \\ \Rightarrow 42 \end{array}$$

Even though this calculation takes two extra steps, it at least gives the same, correct answer. Indeed, an important property of each and every program written in Haskell is that it will always yield the same answer when given the same inputs, regardless of the order chosen to perform the calculations.[3] This is precisely the mathematical definition of a *function*: for the same inputs, it always yields the same output.

On the other hand, the first calculation above required fewer steps than the second, and thus it is said to be more *efficient*. Efficiency in both space (amount of memory used) and time (number of steps executed) is important when searching for solutions to problems. Of course, if the computation returns the wrong answer, efficiency is a moot point. In general, it is best to search first for an elegant (and correct!) solution to a problem, and later refine it for better performance. This strategy is sometimes summarized as "Get it right first!"

The above calculations are fairly trivial, but much more sophisticated computations will be introduced soon enough. For starters, and to introduce the idea of a Haskell function, the arithmetic operations performed in the previous example can be *generalized* by defining a function to perform them for any numbers x, y, and z:

$$\mathit{simple}\ x\ y\ z = x * (y + z)$$

This equation defines *simple* as a function of three *arguments*, x, y, and z. Note the use of *spaces* in this definition to separate the function name, *simple*, from its arguments, x, y, and z. Unlike many other programming languages, Haskell functions are defined by providing first the function name and then any arguments, separated by spaces. More traditional notations for this function would look like this:

$$\begin{array}{l} \mathit{simple}(x, y, z) = x \times (y + z) \\ \mathit{simple}(x, y, z) = x \cdot (y + z) \\ \mathit{simple}(x, y, z) = x(y + z) \\ \mathit{simple}(x, y, z) = x * (y + z) \end{array}$$

[3] This is true as long as a non-terminating sequence of calculations is not chosen, another issue that will be addressed later.

Copyrighted material

Incidentally, the last one is also acceptable Haskell syntax – but it is not interchangeable with the previous Haskell definition. Writing *simple x y z* actually means something very different from writing *simple* (x,y,z) in Haskell. Usage of parentheses around Haskell function arguments indicates a tuple – a concept that will be discussed in more detail later in the text.

In any case, it should be clear that "*simple* 3 9 5" is the same as "$3*(9+5)$," and that the proper way to calculate the result is:

$$\begin{aligned}&\mathit{simple}\ 3\ 9\ 5\\&\Rightarrow 3*(9+5)\\&\Rightarrow 3*14\\&\Rightarrow 42\end{aligned}$$

The first step in this calculation is an example of *unfolding* a function definition: 3 is substituted for x, 9 for y, and 5 for z on the right-hand side of the definition of *simple*. This is an entirely mechanical process, not unlike what the computer actually does to execute the program.

simple 3 9 5 is said to *evaluate* to 42. To express the fact that an expression e evaluates (via zero, one, or possibly many more steps) to the value v, we will write $e \Longrightarrow v$ (this arrow is longer than that used earlier). So we can say directly, for example, that *simple* 3 9 5 $\Longrightarrow$ 42, which should be read "*simple* 3 9 5 evaluates to 42."

With *simple* now suitably defined, we can repeat the sequence of arithmetic calculations as often as we like, using different values for the arguments to *simple*. For example, *simple* 4 3 2 $\Longrightarrow$ 20.

We can also use calculation to *prove properties* about programs. For example, it should be clear that for any a, b, and c, *simple a b c* should yield the same result as *simple a c b*. For a proof of this, we calculate *symbolically* – that is, using the symbols a, b, and c rather than concrete numbers such as 3, 5, and 9:

$$\begin{aligned}&\mathit{simple}\ a\ b\ c\\&\Rightarrow a*(b+c)\\&\Rightarrow a*(c+b)\\&\Rightarrow \mathit{simple}\ a\ c\ b\end{aligned}$$

Note that the same notation is used for these symbolic steps as for concrete ones. In particular, the arrow in the notation reflects the direction of formal reasoning, and nothing more. In general, if $e1 \Rightarrow e2$, then it is also true that $e2 \Rightarrow e1$.

These symbolic steps are also referred to as "calculations," even though the computer will not typically perform them when executing a program (although

Copyrighted material

it might perform them *before* a program is run if it thinks that it might make the program run faster). The second step in the calculation above relies on the commutativity of addition (for any numbers x and y, $x + y = y + x$). The third step is the reverse of an unfold step, and is appropriately called a *fold* calculation. It would be particularly strange if a computer performed this step while executing a program, since it does not seem to be headed toward a final answer. But for proving properties about programs, such "backward reasoning" is quite important.

When we wish to spell out the justification for each step, whether symbolic or concrete, a calculation can be annotated with more detail, as in:

$$\begin{array}{l} \mathit{simple}\ a\ b\ c \\ \ \Rightarrow \{\mathit{unfold}\} \\ a * (b + c) \\ \ \Rightarrow \{\mathit{commutativity}\} \\ a * (c + b) \\ \ \Rightarrow \{\mathit{fold}\} \\ \mathit{simple}\ a\ c\ b \end{array}$$

In most cases, however, this will not be necessary.

Proving properties of programs is another theme that will be repeated often in this text. Computer music applications often have some kind of a mathematical basis, and that mathematics must be reflected somewhere in our programs. But how do we know if we got it right? Proof by calculation is one way to connect the problem specification with the program solution.

More broadly speaking, as the world begins to rely more and more on computers to accomplish not just ordinary tasks such as writing term papers, sending e-mail, and social networking but also life-critical tasks such as controlling medical procedures and guiding spacecraft, the correctness of programs gains in importance. Proving complex properties of large, complex programs is not easy – and rarely, if ever, done in practice – but that should not deter us from proving simpler properties of the whole system, or complex properties of parts of the system, since such proofs may uncover errors, and if not, will at least give us confidence in our effort.

If you are someone who is already an experienced programmer, the idea of computing *everything* by calculation may seem odd at best and naïve at worst. How do we write to a file, play a sound, draw a picture, or respond to mouse-clicks? If you are wondering about these things, it is hoped that you have patience reading the early chapters, and that you find delight in reading the later chapters where the full power of this approach begins to shine.

Copyrighted material

In many ways this first chapter is the most difficult, since it contains the highest density of new concepts. If the reader has trouble with some of the concepts in this overview chapter, keep in mind that most of them will be revisited in later chapters, and do not hesitate to return to this chapter later to reread difficult sections; they will likely be much easier to grasp at that time.

Details: In the remainder of this textbook the need will often arise to explain some aspect of Haskell in more detail, without distracting too much from the primary line of discourse. In those circumstances the explanations will be offset in a shaded box such as this one, proceeded with the word "Details."

Exercise 1.1 Write out all of the steps in the calculation of the value of *simple* (*simple* 2 3 4) 5 6.

Exercise 1.2 Prove by calculation that *simple* $(a - b)\ a\ b \Longrightarrow a^2 - b^2$.

1.4 Expressions and Values

In Haskell, the entities on which calculations are performed are called *expressions*, and the entities that result from a calculation – i.e., "the answers" – are called *values*. It is helpful to think of a value just as an expression on which no more calculation can be carried out – every value is an expression, but not the other way around.

Examples of expressions include *atomic* (meaning indivisible) values such as the integer 42 and the character `'a'`, which are examples of two *primitive* atomic values in Haskell. The next chapter introduces examples of *constructor* atomic values, such as the musical notes *C*, *D*, *Ef*, *Fs*, etc., which in standard music notation are written C, D, E♭, F♯, etc., and are pronounced C, D, E-flat, F-sharp, etc. (In music theory, note names are called *pitch classes*.)

In addition, there are *structured* expressions (i.e., made from smaller pieces) such as the *list* of pitches $[C, D, Ef]$, the character/number *pair* (`'b'`,4) (lists and pairs are different in a subtle way, to be described later), and the string `"Euterpea"`. Each of these structured expressions is also a value, since by themselves there is no further calculation that can be carried out. As another example, $1 + 2$ is an expression, and one step of calculation yields the expression 3, which is a value, since no more calculations can be performed. As a final example, as was explained earlier, the expression *simple* 3 9 5 evaluates to the value 42.

Copyrighted material

Haskell's *type system* ensures that Haskell programs are *well-typed*; that is, that the programmer has not mismatched types in some way. For example, it does not make much sense to add together two characters, so the expression `'a' + 'b'` is *ill-typed*. The best news is that Haskell's type system will tell you if your program is well-typed *before you run it*. This is a big advantage, since most programming errors are manifested as type errors.

1.6 Function Types and Type Signatures

What should the type of a function be? It seems that it should at least convey the fact that a function takes values of one type – say, *T1* – as input, and returns values of (possibly) some other type – say, *T2* – as output. In Haskell this is written *T1* → *T2*, and such a function is said to "map values of type *T1* to values of type *T2*."[5] If there is more than one argument, the notation is extended with more arrows. For example, if the intent is that the function *simple* defined in the previous section has type *Integer* → *Integer* → *Integer* → *Integer*, we can include a type signature with the definition of *simple*:

$$\begin{aligned}&\mathit{simple} && :: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}\\ &\mathit{simple}\ x\ y\ z && = x * (y + z)\end{aligned}$$

Details: When writing Haskell programs using a typical text editor, there will not be nice fonts and arrows as in *Integer* → *Integer*. Rather, you will have to type `Integer -> Integer`.

Haskell's type system also ensures that user-supplied type signatures such as this one are correct. Actually, Haskell's type system is powerful enough to allow us to avoid writing any type signatures at all, in which case the type system is said to *infer* the correct types.[6] Nevertheless, judicious placement of type signatures, as was done for *simple*, is a good habit, since type signatures are an effective form of documentation and help bring programming errors to light. In fact, it is a good habit to first write down the type of each function you are planning to define, as a first approximation to its full specification – a way to grasp its overall functionality before delving into its details.

The normal use of a function is referred to as *function application*. For example, *simple* 3 9 5 is the application of the function *simple* to the arguments 3, 9, and 5. Some functions, such as (+), are applied using what is known as

[5] In mathematics *T1* is called the *domain* of the function and *T2* the *range*.
[6] There are a few exceptions to this rule, and in the case of *simple*, the inferred type is actually a bit more general than that written in this chapter. Both of these points will be returned to later.

Copyrighted material

infix syntax; that is, the function is written between the two arguments rather than in front of them (compare $x + y$ to $f\ x\ y$).

Details: Infix functions are often called *operators*, and are distinguished by the fact that they do not contain any numbers or letters of the alphabet. Thus ^! and $*$# : are infix operators, whereas *thisIsAFunction* and *f9g* are not (but are still valid names for functions or other values). The only exception to this is that the symbol ' is considered to be alphanumeric; thus f' and *one's* are valid names, but not operators.

In Haskell, when referring to an infix operator as a value, it is enclosed in parentheses, such as when declaring its type, as in:

$(+) :: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}$

Also, when trying to understand an expression such as $f\ x + g\ y$, there is a simple rule to remember: function application *always* has "higher precedence" than operator application, so that $f\ x + g\ y$ is the same as $(f\ x) + (g\ y)$.

Despite all of these syntactic differences, however, operators are still just functions.

Exercise 1.3 Identify the well-typed expressions in the following and, for each, give its proper type:

$[A, B, C]$
$[D, 42]$
$(-42, \mathit{Ef})$
[('a', 3), ('b', 5)]
simple 'a' 'b' 'c'
(*simple* 1 2 3, *simple*)
["I","love","Euterpea"]

For those expressions that are ill-typed, explain why.

1.7 Abstraction, Abstraction, Abstraction

The title of this section is the answer to the question "What are the three most important ideas in programming?" Webster defines the verb "abstract" as follows:

Copyrighted material

abstract, *vt* (1) remove, separate (2) to consider apart from application to a particular instance.

In programming this happens when a pattern repeats itself and we wish to "separate" that pattern from the "particular instances" in which it appears. In this textbook this process is called the *abstraction principle*. The following sections introduce several different kinds of abstraction, using examples involving both simple numbers and arithmetic (things everyone should be familiar with) as well as musical examples (that are specific to Euterpea).

1.7.1 Naming

One of the most basic ideas in programming – for that matter, in everyday life – is to *name* things. For example, we may wish to give a name to the value of π, since it is inconvenient to retype (or remember) the value of π beyond a small number of digits. In mathematics the Greek letter π in fact *is* the name for this value, but unfortunately we do not have the luxury of using Greek letters on standard computer keyboards and/or text editors. So in Haskell we write:

```
pi :: Double
pi = 3.141592653589793
```

to associate the name *pi* with the number 3.141592653589793. The type signature in the first line declares *pi* to be a *double-precision floating-point number*, which mathematically and in Haskell is distinct from an integer.[7] Now the name *pi* can be used in expressions whenever it is in scope; it is an abstract representation, if you will, of the number 3.141592653589793. Furthermore, if there is ever a need to change a named value (which hopefully will not ever happen for *pi*, but could certainly happen for other values), we would only have to change it in one place, instead of in the possibly large number of places where it is used.

For a simple musical example, note first that in music theory, a *pitch* consists of a *pitch class* and an *octave*. For example, in Euterpea we simply write $(A, 4)$ to represent the pitch class A in the fourth octave. This particular note is called "concert A" (because it is often used as the note to which an orchestra tunes its instruments) or "A440" (because its frequency is 440 cycles per second). Because this particular pitch is so common, it may be desirable to give it a name, which is easily done in Haskell, as was done above for π:

[7] We will have more to say about floating-point numbers later.

Copyrighted material

```
concertA, a440 :: (PitchClass, Octave)
concertA = (A, 4)   -- concert A
a440     = (A, 4)   -- A440
```

> **Details:** This example demonstrates the use of program *comments*. Any text to the right of "- -" till the end of the line is considered to be a programmer comment and is effectively ignored. Haskell also permits *nested* comments that have the form {-this is a comment -} and can appear anywhere in a program, including across multiple lines.

This example demonstrates the (perhaps obvious) fact that several different names can be given to the same value – just as your brother John might have the nickname "Moose." Also note that the name *concertA* requires more typing than $(A, 4)$; nevertheless, it has more mnemonic value and, if mistyped, will more likely result in a syntax error. For example, if you type "*concrtA*" by mistake, you will likely get an error saying, "Undefined variable," whereas if you type "$(A, 5)$," you will not.

> **Details:** This example also demonstrates that two names having the same type can be combined into the same type signature, separated by a comma. Note finally, as a reminder, that these are names of values, and thus they both begin with a lowercase letter.

Consider now a problem whose solution requires writing some larger expression more than once. For example:

```
x :: Float
x = f (pi * r ** 2) + g (pi * r ** 2)
```

> **Details:** $(**)$ is Haskell's floating-point exponentiation operator. Thus $pi * r ** 2$ is analogous to πr^2 in mathematics. $(**)$ has higher precedence than $(*)$ and the other binary arithmetic operators in Haskell.

Note in the definition of x that the expression $pi * r ** 2$ (presumably representing the area of a circle whose radius is r) is repeated – it has two instances – and thus, applying the abstraction principle, it can be separated

Copyrighted material

from these instances. From the previous examples, doing this is straightforward – it is called *naming* – so we might choose to rewrite the single equation above as two:

$$\begin{aligned} &area &&= pi * r ** 2 \\ &x &&= f\ area + g\ area \end{aligned}$$

If, however, the definition of *area* is not intended for use elsewhere in the program, then it is advantageous to "hide" it within the definition of x. This will avoid cluttering up the namespace, and prevents *area* from clashing with some other value named *area*. To achieve this, we could simply use a **let** expression:

$$\begin{aligned} x = &\ \mathbf{let}\ area = pi * r ** 2 \\ &\ \mathbf{in}\ f\ area + g\ area \end{aligned}$$

A **let** expression restricts the *visibility* of the names that it creates to the internal workings of the **let** expression itself. For example, if we were to write:

$$\begin{aligned} &area &&= 42 \\ &x &&= \mathbf{let}\ area = pi * r ** 2 \\ & && \quad\ \mathbf{in}\ f\ area + g\ area \end{aligned}$$

then there is no conflict of names – the "outer" *area* is completely different from the "inner" one enclosed in the **let** expression. Think of the inner *area* as analogous to the first name of someone in your household. If your brother's name is John, he will not be confused with John Thompson who lives down the street when you say, "John spilled the milk."

So you can see that naming – using either top-level equations or equations within a **let** expression – is an example of the abstraction principle in action.

Details: An equation such as $c = 42$ is called a *binding*. A simple rule to remember when programming in Haskell is never to give more than one binding for the same name in a context where the names can be confused, whether at the top level of your program or nestled within a **let** expression. For example, this is not allowed:

$$\begin{aligned} a &= 42 \\ a &= 43 \end{aligned}$$

nor is this:

$$\begin{aligned} a &= 42 \\ b &= 43 \\ a &= 44 \end{aligned}$$

Copyrighted material

```
mel :: Music Pitch
mel = hNote qn p1 :+: hNote qn p2 :+: hNote qn p3
```

Again using the idea of unfolding described earlier in this chapter, it is easy to prove that this definition is equivalent to the previous one.

As with *areaF*, this use of *hNote* is an example of functional abstraction. In a sense, functional abstraction can be seen as a generalization of naming. That is, *area r1* is just a name for *pi* ∗ *r1* ∗∗ 2, *hNote d p* is just a name for *note d p* :=: *note d* (*trans* (−3) *p*), and so on. Stated another way, named quantities such as *area*, *pi*, *concertA*, and *a440* defined earlier can be thought of as functions with no arguments.

Of course, the definition of *hNote* could also be hidden within *mel* using a **let** expression as was done in the previous example:

```
mel :: Music Pitch
mel = let hNote d p = note d p :=: note d (trans (−3) p)
      in hNote qn p1 :+: hNote qn p2 :+: hNote qn p3
```

1.7.3 Data Abstraction

The value of *mel* is the sequential composition of three harmonized notes. But what if in another situation we must compose together five harmonized notes, or in other situations even more? In situations where the number of values is uncertain, it is useful to represent them in a *data structure*. For the example at hand, a good choice of data structure is a *list*, briefly introduced earlier, that can have any length. The use of a data structure motivated by the abstraction principle is one form of *data abstraction*.

Imagine now an entire list of pitches whose length is not known at the time the program is written. What now? It seems that a function is needed to convert a list of pitches into a sequential composition of harmonized notes. Before defining such a function, however, there is a bit more to say about lists.

Earlier the example [*C*, *D*, *Ef*] was given, a list of pitch classes whose type is thus [*PitchClass*]. A list with *no* elements is – not surprisingly – written [ ], and is called the *empty list*.

To add a single element *x* to the front of a list *xs*, we write *x* : *xs* in Haskell. (Note the naming convention used here: *xs* is the plural of *x*, and should be read that way.) For example, *C* : [*D*, *Ef*] is the same as [*C*, *D*, *Ef*]. In fact, this list is equivalent to *C* : (*D* : (*Ef* : [ ])), which can also be written *C* : *D* : *Ef* : [ ], since the infix operator (:) is right-associative.

Copyrighted material

Details: In mathematics we rarely worry about whether the notation $a + b + c$ stands for $(a + b) + c$ (in which case $+$ would be "left-associative") or $a + (b + c)$ (in which case $+$ would be "right-associative"). This is because, in situations where the parentheses are left out, it is usually the case that the operator is *mathematically* associative, meaning that it does not matter which interpretation is chosen. If the interpretation *does* matter, mathematicians will include parentheses to make it clear. Furthermore, in mathematics there is an implicit assumption that some operators have higher *precedence* than others; for example, $2 \times a + b$ is interpreted as $(2 \times a) + b$, not $2 \times (a + b)$.

In many programming languages, including Haskell, each operator is defined to have a particular precedence level and to be left-associative or right-associative, or to have no associativity at all. For arithmetic operators, mathematical convention is usually followed; for example, $2 * a + b$ is interpreted as $(2 * a) + b$ in Haskell. The predefined list-forming operator (:) is defined to be right-associative. Just as in mathematics, this associativity can be overridden by using parentheses: thus $(a : b) : c$ is a valid Haskell expression (assuming that it is well-typed; it must be a list of lists), and is very different from $a : b : c$. A way to specify the precedence and associativity of user-defined operators will be discussed in a later chapter.

Returning now to the problem of defining a function (call it *hList*) to turn a list of pitches into a sequential composition of harmonized notes, we should first express what its type should be:

$$hList :: Dur \rightarrow [Pitch] \rightarrow Music\ Pitch$$

To define its proper behavior, it is helpful to consider, one by one, all possible cases that could arise on the input. First off, the list could be empty, in which case the sequential composition should be a *Music Pitch* value that has zero duration. So:

$$hList\ d\ [\,] = rest\ 0$$

The other possibility is that the list *is not* empty – i.e., it contains at least one element, say *p*, followed by the rest of the elements, say *ps*. In this case the result should be the harmonization of *p* followed by the sequential composition of the harmonization of *ps*. Thus:

$$hList\ d\ (p : ps) = hNote\ d\ p :\!+\!:\ hList\ d\ ps$$

Copyrighted material

Note that this part of the definition of *hList* is *recursive* – it refers to itself! But the original problem – the harmonization of $p : ps$ – has been reduced to the harmonization of p (previously captured in the function *hNote*) and the harmonization of *ps* (a slightly smaller problem than the original one).

Combining these two equations with the type signature yields the complete definition of the function *hList*:

```
hList            :: Dur → [Pitch] → Music Pitch
hList d [ ]       = rest 0
hList d (p : ps) = hNote d p :+: hList d ps
```

Recursion is a powerful technique that will be used many times in this textbook. It is also an example of a general problem-solving technique where a large problem is broken down into several smaller but similar problems; solving these smaller problems one by one leads to a solution to the larger problem.

Details: Although intuitive, this example highlights an important aspect of Haskell: *pattern matching*. The left-hand sides of the equations contain *patterns* such as [ ] and $x : xs$. When a function is applied, these patterns are *matched* against the argument values in a fairly intuitive way ([ ] only matches the empty list, and $p : ps$ will successfully match any list with at least one element, while naming the first element p and the rest of the list *ps*). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results. All of the equations that define a particular function must appear together, one after the other.

Defining functions by pattern matching is quite common in Haskell, and you should eventually become familiar with the various kinds of patterns that are allowed; see Appendix D for a concise summary.

Given this definition of *hList* the definition of *mel* can be rewritten as:

```
mel = hList qn [p1, p2, p3]
```

We can prove that this definition is equivalent to the old one via calculation:

```
mel = hList qn [p1, p2, p3]
 ⇒ hList qn (p1 : p2 : p3 : [ ])
 ⇒ hNote qn p1 :+: hList qn (p2 : p3 : [ ])
 ⇒ hNote qn p1 :+: hNote qn p2 :+: hList qn (p3 : [ ])
 ⇒ hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: hList qn [ ]
 ⇒ hNote qn p1 :+: hNote qn p2 :+: hNote qn p3 :+: rest 0
```

Copyrighted material

The first step above is not really a calculation, but rather a rewriting of the list syntax. The remaining calculations each represent an unfolding of *hList*.

Lists are perhaps the most commonly used data structure in Haskell, and there is a rich library of functions that operate on them. In subsequent chapters, lists will be used in a variety of interesting computer music applications.

Exercise 1.4 Modify the definitions of *hNote* and *hList* so that they each take an extra argument that specifies the interval of harmonization (rather than being fixed at -3). Rewrite the definition of *mel* to take these changes into account.

1.8 Haskell Equality versus Musical Equality

The astute reader will have objected to the proof just completed, arguing that the original version of *mel*:

hNote qn p1 :+: *hNote qn p2* :+: *hNote qn p3*

is not the same as the terminus of the above proof:

hNote qn p1 :+: *hNote qn p2* :+: *hNote qn p3* :+: *rest* 0

Indeed, that reader would be right! As Haskell values, these expressions are *not* equal, and if you printed each of them you would get different results. So what happened? Did proof by calculation fail?

No, proof by calculation did not fail, since, as just pointed out, as Haskell values, these two expressions are not the same, and proof by calculation is based on the equality of Haskell values. The problem is that a "deeper" notion of equivalence is needed, one based on the notion of *musical* equality. Adding a rest of zero duration to the beginning or end of any piece of music should not change what we *hear*, and therefore it seems that the above two expressions are *musically* equivalent. But it is unreasonable to expect Haskell to figure this out for the programmer!

As an analogy, consider the use of an ordered list to represent a set (which is unordered). The Haskell values $[x1, x2]$ and $[x2, x1]$ are not equal, yet in a program that "interprets" them as sets, they *are* equal.

The way this problem is approached in Euterpea is to formally define a notion of *musical interpretation*, from which the notion of *musical equivalence* is defined. This leads to a kind of "algebra of music" that includes, among others, the following axiom:

m :+: *rest* $0 \equiv m$

Copyrighted material

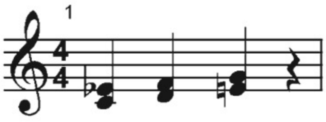

Figure 1.1 Polyphonic versus contrapuntal interpretation.

The operator (≡) should be read "is musically equivalent to." With this axiom it is easy to see that the original two expressions above *are* in fact musically equivalent.

For a more extreme example of this idea, and to entice the reader to learn more about musical equivalence in later chapters, note that *mel*, given pitches $p1 = Ef, p2 = F, p3 = G$ and duration $d = 1/4$, generates the harmonized melody shown in Figure 1.1; we can write this concretely in Euterpea as:

```
mel1 = (note (1/4) (Ef, 4) :=: note (1/4) (C, 4))  :+:
       (note (1/4) (F,  4) :=: note (1/4) (D, 4))  :+:
       (note (1/4) (G,  4) :=: note (1/4) (E, 4))
```

The definition of *mel1* can then be seen as a *polyphonic* interpretation of the musical phrase in Figure 1.1, where each pair of notes is seen as a harmonic unit. In contrast, a *contrapuntal* interpretation sees two independent *lines*, or *voices*, in this case the line ⟨E♭,F,G⟩ and the line ⟨C,D,E⟩. In Euterpea we can write this as:

```
mel2 = (note (1/4) (Ef, 4) :+: note (1/4) (F, 4) :+: note (1/4) (G, 4))
        :=:
       (note (1/4) (C,  4) :+: note (1/4) (D, 4) :+: note (1/4) (E, 4))
```

mel1 and *mel2* are clearly not equal as Haskell values. Yet if they are played, they will *sound* the same – they are, in the sense described earlier, *musically equivalent*. But proving these two phrases musically equivalent will require far more than a simple axiom involving *rest* 0. In fact, this can be done in an elegant way using the algebra of music developed in Chapter 12.

1.9 Code Reuse and Modularity

There does not seem to be much repetition in the last definition of *hList*, so perhaps the end of the abstraction process has been reached. In fact, it is worth considering how much progress has been made. The original definition:

Copyrighted material

happen when the magnitude of $b + c$ differs significantly from the magnitude of either b or c. For example, the following two calculations are from GHC:

$$5 * (-0.123456 + 0.123457) :: \mathit{Float} \Rightarrow 4.991889\text{e}{-6}$$
$$5 * (-0.123456) + 5 * (0.123457) :: \mathit{Float} \Rightarrow 5.00679\text{e}{-6}$$

Although the discrepancy here is small, its very existence is worrisome, and in certain situations it could be disastrous. The precise behavior of floating-point numbers will not be discussed further in this textbook. Just remember that they are *approximations* to the real numbers. If real-number accuracy is important to your application, further study of the nature of floating-point numbers is probably warranted.

On the other hand, the distributive law (and many others) is valid in Haskell for the exact data types *Integer* and *Ratio Integer* (i.e., rationals). Although the representation of an *Integer* in Haskell is not normally something to be concerned about, it should be clear that the representation must be allowed to grow to an arbitrary size. For example, Haskell has no problem with the following number:

```
veryBigNumber :: Integer
veryBigNumber = 43208345720348593219876512372134059
```

and such numbers can be added, multiplied, etc., without any loss of accuracy. However, such numbers cannot fit into a single word of computer memory, most of which is limited to 32 or 64 bits. Worse, since the computer system does not know ahead of time exactly how many words will be required, it must devise a dynamic scheme to allow just the right number of words to be used in each case. The overhead of implementing this idea unfortunately causes programs to run slower.

For this reason, Haskell (and most other languages) provides another integer data type, called *Int*, that has maximum and minimum values that depend on the word size of the particular computer being used. In other words, every value of type *Int* fits into one word of memory, and the primitive machine instructions for binary numbers can be used to manipulate them efficiently.[9] Unfortunately, this means that *overflow* or *underflow* errors could occur when an *Int* value exceeds either the maximum or minimum value. Sadly, most implementations of Haskell (as well as most other languages) do not tell you when this happens. For example, in GHC running on a 32-bit processor, the following *Int* value:

[9] The Haskell Report requires that every implementation support *Int*s at least in the range -2^{29} to $2^{29} - 1$, inclusive. The GHC implementation running on a 32-bit processor, for example, supports the range -2^{31} to $2^{31} - 1$.

Copyrighted material

```
i :: Int
i = 1234567890
```

works just fine, but if you multiply it by two, GHC returns the value -1825831516! This is because twice *i* exceeds the maximum allowed value, so the resulting bits become nonsensical,[10] and are interpreted in this case as a negative number of the given magnitude.

This is alarming! Indeed, why should anyone ever use *Int* when *Integer* is available? The answer, as implied earlier, is efficiency, but clearly care should be taken when making this choice. If you are indexing into a list, for example, and you are confident that you are not performing index calculations that might result in the above kind of error, then *Int* should work just fine, since a list longer than 2^{31} will not fit into memory anyway! But if you are calculating the number of microseconds in some large time interval or counting the number of people living on earth, then *Integer* would most likely be a better choice. Choose your number data types wisely!

In this textbook the numeric data types *Integer*, *Int*, *Float*, *Double*, *Rational*, and *Complex* will be used for a variety of different applications; for a discussion of the other number types, consult the Haskell Report. As these data types are used, there will be little discussion about their properties – this is not, after all, a book on numerical analysis – but a warning will be cast whenever reasoning about, for example, floating-point numbers in a way that might not be technically sound.

[10] Actually, these bits are perfectly sensible in the following way: the 32-bit binary representation of *i* is 01001001100101100000001011010010, and twice that is 10010011001011000000010110100100. But the latter number is seen as negative, because the 32nd bit (the highest-order bit on the CPU on which this was run) is a one, which means it is a negative number in "twos-complement" representation. The twos-complement of this number is in turn 01101100110100111111101001011100, whose decimal representation is 1825831516.

Copyrighted material

2
Simple Music

The previous chapters introduced some of the fundamental ideas of functional programming in Haskell. Also introduced were several of Euterpea's functions and operators, such as *note*, *rest*, (:+:), (:=:), and *trans*. This chapter will reveal the actual definitions of these functions and operators, thus exposing Euterpea's underlying structure and overall design at the note level. In addition, a number of other musical ideas will be developed and, in the process, more Haskell features will be introduced as well.

2.1 Preliminaries

Sometimes it is convenient to use a built-in Haskell data type to directly represent some concept of interest. For example, we may wish to use *Int* to represent *octaves*, where by convention octave 4 corresponds to the octave containing middle C on the piano. We can express this in Haskell using a *type synonym*:

```
type Octave = Int
```

A type synonym does not create a new data type, it just gives a new name to an existing type. Type synonyms can be defined not just for atomic types such as *Int*, but also for structured types such as pairs. For example, as discussed in the last chapter, in music theory a pitch is defined as a pair consisting of a *pitch class* and an *octave*. Assuming the existence of a data type called *PitchClass* (which we will return to shortly), we can write the following type synonym:

```
type Pitch = (PitchClass, Octave)
```

For example, concert A (i.e., A440) corresponds to the pitch $(A, 4)$ :: *Pitch*, and the lowest and highest notes on a piano correspond to $(A, 0)$ :: *Pitch* and $(C, 8)$ :: *Pitch*, respectively.

Copyrighted material

Another important musical concept is *duration*. Rather than using either integers or floating-point numbers, Euterpea uses *rational* numbers to denote duration:

```
type Dur = Rational
```

Rational is the data type of rational numbers expressed as ratios of *Integer*s in Haskell. The choice of *Rational* is somewhat subjective, but is justified by three observations: (1) many durations are expressed as ratios in music theory (5:4 rhythm, quarter notes, dotted notes, and so on), (2) *Rational* numbers are exact (unlike floating point numbers), which is important in many computer music applications, and (3) irrational durations are rarely needed.

Rational numbers in Haskell are printed by GHC in the form *n* % *d*, where *n* is the numerator and *d* is the denominator. Even a whole number, say the number 42, will print as 42 % 1 if it is a *Rational* number. To create a *Rational* number in a program, however, once it is given the proper type, we can use the normal division operator, as in the following definition of a quarter note:

```
qn :: Dur
qn = 1/4   -- quarter note
```

So far, so good. But what about *PitchClass*? We might try to use integers to represent pitch classes as well, but this is not very elegant. Ideally, we would like to write something that looks more like the conventional pitch class names C, C♯, D♭, D, etc. The solution is to use an *algebraic data type* in Haskell:

```
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Ff | Es | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
```

Details: All of the names to the right of the equal sign in a **data** declaration are called *constructors*, and must be capitalized. In this way they are syntactically distinguished from ordinary values. This distinction is useful, since only constructors can be used in the pattern matching that is part of a function definition, as will be described shortly.

The *PitchClass* data type declaration essentially enumerates 35 pitch class names (five for each of the note names A through G). Note that both double-sharps and double-flats are included, resulting in many enharmonics (i.e., two notes that "sound the same," such as G♯ and A♭).

Copyrighted material

(The order of the pitch classes may seem a bit odd, but the idea is that if a pitch class *pc1* is to the left of a pitch class *pc2*, then *pc1*'s pitch is "lower than" *pc2*'s. This idea will be formalized and exploited in Chapter 7.1.)

Keep in mind that *PitchClass* is a completely new, user-defined data type that is not equal to any other. This is what distinguishes a **data** declaration from a **type** declaration. As another example of the use of a **data** declaration to define a simple enumerated type, Haskell's Boolean data type, called *Bool*, is predefined in Haskell simply as:

```
data Bool = False | True
```

2.2 Notes, Music, and Polymorphism

We can, of course, define other data types for other purposes. For example, we will want to define the notion of a *note* and a *rest*. Both of these can be thought of as "primitive" musical values, and thus as a first attempt we might write:

```
data Primitive = Note Dur Pitch
               | Rest Dur
```

Analogous to our previous data type declarations, the above declaration says that a *Primitive* is either a *Note* or a *Rest*. However, it is different in that the constructors *Note* and *Rest* take arguments, like functions do. In the case of *Note*, it takes two arguments, whose types are *Dur* and *Pitch*, whereas *Rest* takes one argument, a value of type *Dur*. In other words, the types of *Note* and *Rest* are:

```
Note :: Dur → Pitch → Primitive
Rest :: Dur →         Primitive
```

For example, *Note qn a440* is concert A played as a quarter note, and *Rest* 1 is a whole-note rest.

This definition is not completely satisfactory, however, because we may wish to attach other information to a note, such as its loudness, or some other annotation or articulation. Furthermore, the pitch itself may actually be a percussive sound, having no true pitch at all. To resolve this, Euterpea uses an important concept in Haskell, namely *polymorphism* – the ability to parameterize, or abstract, over types (*poly* means *many* and *morphism* refers to the structure, or *form*, of objects).

Primitive can be redefined as a *polymorphic data type* as follows. Instead of fixing the type of the pitch of a note, it is left unspecified through the use of a *type variable*:

Copyrighted material

(Recall that these last two operators were introduced in the last chapter. You can see now that they are actually constructors of an algebraic data type.)

- *Modify cntrl m* is an "annotated" version of *m* in which the control parameter *cntrl* specifies some way *m* is to be modified.

> **Details:** Note that *Music a* is defined in terms of *Music a*, and thus the data type is said to be *recursive* (analogous to a recursive function). It is also often called an *inductive* data type, since it is, in essence, an inductive definition of an infinite number of values, each of which can be arbitrarily complex.

It is convenient to represent these musical ideas as a recursive data type, because it allows us to not only *construct* musical values, but also take them apart, analyze their structure, print them in a structure-preserving way, transform them, interpret them for performance purposes, and so on. Many examples of these kinds of processes will be seen in this textbook.

The *Control* data type is used by the *Modify* constructor to annotate a *Music* value with a *tempo change*, a *transposition*, a *phrase attribute*, an *instrument*, a *key signature*, or a *custom label*. This data type is unimportant at the moment, but for completeness here is its full definition:

```
data Control =
      Tempo      Rational            -- scale the tempo
    | Transpose  AbsPitch            -- transposition
    | Instrument InstrumentName      -- instrument label
    | Phrase     [PhraseAttribute]   -- phrase attributes
    | KeySig     PitchClass Mode     -- key signature and mode
    | Custom     String              -- custom label
data Mode = Major | Minor | Ionian | Dorian | Phrygian
          | Lydian | Mixolydian | Aeolian | Locrian
          | CustomMode String
```

AbsPitch ("absolute pitch," to be defined in Section 2.4) is just a type synonym for *Int*. Instrument names are borrowed from the General MIDI standard [3, 4], and are captured as an algebraic data type in Figure 2.1. The *KeySig* constructor attaches a key signature to a *Music* value and is different from transposition. A full explanation of phrase attributes and the custom labels is deferred until Chapter 9.

Copyrighted material

```
data InstrumentName =
    AcousticGranDPiano   | BrightAcousticPiano | ElectricGrandPiano
  | HonkyTonkPiano       | RhodesPiano         | ChorusedPiano
  | Harpsichord          | Clavinet            | Celesta
  | Glockenspiel         | MusicBox            | Vibraphone
  | Marimba              | Xylophone           | TubularBells
  | Dulcimer             | HammondOrgan        | PercussiveOrgan
  | RockOrgan            | ChurchOrgan         | ReedOrgan
  | Accordion            | Harmonica           | TangoAccordion
  | AcousticGuitarNylon  | AcousticGuitarSteel | ElectricGuitarJazz
  | ElectricGuitarClean  | ElectricGuitarMuted | OverdrivenGuitar
  | DistortionGuitar     | GuitarHarmonics     | AcousticBass
  | ElectricBassFingered | ElectricBassPicked  | FretlessBass
  | SlapBass1            | SlapBass2           | SynthBass1
  | SynthBass2           | Violin              | Viola
  | Cello                | Contrabass          | TremoloStrings
  | PizzicatoStrings     | OrchestralHarp      | Timpani
  | StringEnsemble1      | StringEnsemble2     | SynthStrings1
  | SynthStrings2        | ChoirAahs           | VoiceOohs
  | SynthVoice           | OrchestraHit        | Trumpet
  | Trombone             | Tuba                | MutedTrumpet
  | FrenchHorn           | BrassSection        | SynthBrass1
  | SynthBrass2          | SopranoSax          | AltoSax
  | TenorSax             | BaritoneSax         | Oboe
  | Bassoon              | EnglishHorn         | Clarinet
  | Piccolo              | Flute               | Recorder
  | PanFlute             | BlownBottle         | Shakuhachi
  | Whistle              | Ocarina             | Lead1Square
  | Lead2Sawtooth        | Lead3Calliope       | Lead4Chiff
  | Lead5Charang         | Lead6Voice          | Lead7Fifths
  | Lead8BassLead        | Pad1NewAge          | Pad2Warm
  | Pad3Polysynth        | Pad4Choir           | Pad5Bowed
  | Pad6Metallic         | Pad7Halo            | Pad8Sweep
  | FX1Train             | FX2Soundtrack       | FX3Crystal
  | FX4Atmosphere        | FX5Brightness       | FX6Goblins
  | FX7Echoes            | FX8SciFi            | Sitar
  | Banjo                | Shamisen            | Koto
  | Kalimba              | Bagpipe             | Fiddle
  | Shanai               | TinkleBell          | Agogo
  | SteelDrums           | Woodblock           | TaikoDrum
  | MelodicDrum          | SynthDrum           | ReverseCymbal
  | GuitarFretNoise      | BreathNoise         | Seashore
  | BirdTweet            | TelephoneRing       | Helicopter
  | Applause             | Gunshot             | Percussion
  | CustomInstrument String
```

Figure 2.1 General MIDI instrument names.

Copyrighted material

2.3 Convenient Auxiliary Functions

For convenience, and in anticipation of their frequent use, a number of functions are defined in Euterpea to make it easier to write certain kinds of musical values. For starters:

```
note             :: Dur → a → Music a
note d p         = Prim (Note d p)
rest             :: Dur → Music a
rest d           = Prim (Rest d)
tempo            :: Dur → Music a → Music a
tempo r m        = Modify (Tempo r) m
transpose        :: AbsPitch → Music a → Music a
transpose i m    = Modify (Transpose i) m
instrument       :: InstrumentName → Music a → Music a
instrument i m   = Modify (Instrument i) m
phrase           :: [PhraseAttribute] → Music a → Music a
phrase pa m      = Modify (Phrase pa) m
keysig           :: PitchClass → Mode → Music a → Music a
keysig pc mo m = Modify (KeySig pc mo) m
```

Note that each of these functions is polymorphic, a trait inherited from the data types that it uses. Also recall that the first two of these functions were used in an example in the last chapter.

We can also create simple names for familiar notes, durations, and rests, as shown in Figures 2.2 and 2.3. Despite the large number of them, these names are sufficiently "unusual" that name clashes are unlikely.

Details: Figures 2.2 and 2.3 demonstrate that at the top level of a program, more than one equation can be placed on one line, as long as they are separated by semicolons. This allows us to save vertical space on the page, and is useful whenever each line is relatively short. A semicolon is not needed at the end of a single equation, or at the end of the last equation on a line. This convenient feature is part of Haskell's *layout* rule, and will be explained in more detail later.

More than one equation can also be placed on one line in a **let** expression, as demonstrated in the following:

```
let x = 1; y = 2
in x + y
```

Copyrighted material

```
cff, cf, c, cs, css, dff, df, d, ds, dss, eff, ef, e, es, ess, fff, ff, f,
  fs, fss, gff, gf, g, gs, gss, aff, af, a, as, ass, bff, bf, b, bs, bss ::
    Octave → Dur → Music Pitch
cff o d = note d (Cff, o); cf  o d = note d (Cf,  o)
c   o d = note d (C,   o); cs  o d = note d (Cs,  o)
css o d = note d (Css, o); dff o d = note d (Dff, o)
df  o d = note d (Df,  o); d   o d = note d (D,   o)
ds  o d = note d (Ds,  o); dss o d = note d (Dss, o)
eff o d = note d (Eff, o); ef  o d = note d (Ef,  o)
e   o d = note d (E,   o); es  o d = note d (Es,  o)
ess o d = note d (Ess, o); fff o d = note d (Fff, o)
ff  o d = note d (Ff,  o); f   o d = note d (F,   o)
fs  o d = note d (Fs,  o); fss o d = note d (Fss, o)
gff o d = note d (Gff, o); gf  o d = note d (Gf,  o)
g   o d = note d (G,   o); gs  o d = note d (Gs,  o)
gss o d = note d (Gss, o); aff o d = note d (Aff, o)
af  o d = note d (Af,  o); a   o d = note d (A,   o)
as  o d = note d (As,  o); ass o d = note d (Ass, o)
bff o d = note d (Bff, o); bf  o d = note d (Bf,  o)
b   o d = note d (B,   o); bs  o d = note d (Bs,  o)
bss o d = note d (Bss, o)
```

Figure 2.2 Convenient shorthand for creating *Note* values. The standard adopted is that "x sharp" is *xs*, "x double sharp" is *xss*, "x flat" is *xf*, and "x double flat" is *xff*. This is quite convenient for the vast majority of notes, with one caveat for those who are more musically inclined: *ff* means "f flat" and *fff* means "f double flat" – these names refer to *Note* values, not to the dynamic or loudness values double forte and triple forte, which are often written using those abbreviations on musical scores. Fortunately, the use of these notes is rather rare, and typically their enharmonic equivalents are used instead (*e* and *ef*, respectively).

2.3.1 A Simple Example

As a simple example, suppose we wish to generate a ii-V-I chord progression in a particular major key. In music theory, such a chord progression begins with a minor chord on the second degree of the major scale, followed by a major chord on the fifth degree, and ends in a major chord on the first degree. We can write this in Euterpea, using triads in the key of C major, as follows:

```
t251 :: Music Pitch
t251 = let dMinor = d 4 wn :=: f 4 wn :=: a 4 wn
           gMajor = g 4 wn :=: b 4 wn :=: d 5 wn
           cMajor = c 4 bn :=: e 4 bn :=: g 4 bn
       in dMinor :+: gMajor :+: cMajor
```

Copyrighted material

```
bn, wn, hn, qn, en, sn, tn, sfn, dwn, dhn,
  dqn, den, dsn, dtn, ddhn, ddqn, dden :: Dur
bnr, wnr, hnr, qnr, enr, snr, tnr, sfnr, dwnr, dhnr,
  dqnr, denr, dsnr, dtnr, ddhnr, ddqnr, ddenr :: Music Pitch
bn   = 2;    bnr   = rest bn    -- brevis or double whole note rest
wn   = 1;    wnr   = rest wn    -- whole note rest
hn   = 1/2;  hnr   = rest hn    -- half note rest
qn   = 1/4;  qnr   = rest qn    -- quarter note rest
en   = 1/8;  enr   = rest en    -- eighth note rest
sn   = 1/16; snr   = rest sn    -- sixteenth note rest
tn   = 1/32; tnr   = rest tn    -- thirty-second note rest
sfn  = 1/64; sfnr  = rest sfn   -- sixty-fourth note rest
dwn  = 3/2;  dwnr  = rest dwn   -- dotted whole note rest
dhn  = 3/4;  dhnr  = rest dhn   -- dotted half note rest
dqn  = 3/8;  dqnr  = rest dqn   -- dotted quarter note rest
den  = 3/16; denr  = rest den   -- dotted eighth note rest
dsn  = 3/32; dsnr  = rest dsn   -- dotted sixteenth note rest
dtn  = 3/64; dtnr  = rest dtn   -- dotted thirty-second note rest
ddhn = 7/8;  ddhnr = rest ddhn  -- double-dotted half note rest
ddqn = 7/16; ddqnr = rest ddqn  -- double-dotted quarter note rest
dden = 7/32; ddenr = rest dden  -- double-dotted eighth note rest
```

Figure 2.3 Convenient shorthand for creating *Duration* and *Rest* values. Notice that these adhere closely to the American English standard for naming durations, with the exception of *bn* and *bnr*. This design choice was made because the dotted whole note already occupies the *dwn* and *dwnr* names.

Details: Note that more than one equation is allowed in a **let** expression, just like at the top level of a program. The first characters of each equation, however, must line up vertically, and if an equation takes more than one line, then the subsequent lines must be to the right of the first characters. For example, this is legal:

```
let a = aLongName
          + anEvenLongerName
    b = 56
in ...
```

but neither of these is:

```
let   a = aLongName
    + anEvenLongerName
      b = 56
in ...
```

```
let a = aLongName
           + anEvenLongerName
  b = 56
in ...
```

2.4 Absolute Pitches

Treating pitches simply as integers is useful in many settings, so Euterpea uses a type synonym to define the concept of an "absolute pitch":

```
type AbsPitch = Int
```

The absolute pitch of a (relative) pitch can be defined mathematically as 12 times the octave plus the index of the pitch class. We can express this in Haskell as follows:

```
absPitch            :: Pitch → AbsPitch
absPitch (pc, oct) = 12 ∗ (oct + 1) + pcToInt pc
```

Details: Note the use of pattern matching to match the argument of *absPitch* to a pair.

pcToInt is a function that converts a particular pitch class to an index, easily but tediously expressed, as shown in Figure 2.4. But there is a subtlety: according to music theory convention, pitches are assigned integers in the range 0–11, i.e., modulo 12, starting on pitch class C. In other words, the index of C is 0, C♭ is 11, and B♯ is 0. However, that would mean the absolute pitch of $(C, 4)$, say, would be 60, whereas $(Cf, 4)$ would be 71. Somehow the latter does

```
pcToInt :: PitchClass → Int
pcToInt Cff  = −2; pcToInt Dff  = 0; pcToInt Eff  = 2
pcToInt Cf   = −1; pcToInt Df   = 1; pcToInt Ef   = 3
pcToInt C    = 0;  pcToInt D    = 2; pcToInt E    = 4
pcToInt Cs   = 1;  pcToInt Ds   = 3; pcToInt Es   = 5
pcToInt Css  = 2;  pcToInt Dss  = 4; pcToInt Ess  = 6

pcToInt Fff  = 3;  pcToInt Gff  = 5; pcToInt Aff  = 7
pcToInt Ff   = 4;  pcToInt Gf   = 6; pcToInt Af   = 8
pcToInt F    = 5;  pcToInt G    = 7; pcToInt A    = 9
pcToInt Fs   = 6;  pcToInt Gs   = 8; pcToInt As   = 10
pcToInt Fss  = 7;  pcToInt Gss  = 9; pcToInt Ass  = 11

pcToInt Bff  = 9
pcToInt Bf   = 10
pcToInt B    = 11
pcToInt Bs   = 12
pcToInt Bss  = 13
```

Figure 2.4 Converting pitch classes to integers.

Copyrighted material

not seem right; 59 would be a more logical choice. Therefore, the definition in Figure 2.4 is written in such a way that the wrap-around does not happen, i.e., numbers outside the range 0–11 are used. With this definition, *absPitch* (*Cf*, 4) yields 59, as desired.

Details: The repetition of "*pcToInt*" above can be avoided by using a Haskell **case** expression, resulting in a more compact definition:

```
pcToInt    :: PitchClass → Int
pcToInt pc = case pc of
   Cff → −2; Cf → −1; C → 0;  Cs → 1;  Css → 2;
   Dff → 0;  Df → 1;  D → 2;  Ds → 3;  Dss → 4;
   Eff → 2;  Ef → 3;  E → 4;  Es → 5;  Ess → 6;
   Fff → 3;  Ff → 4;  F → 5;  Fs → 6;  Fss → 7;
   Gff → 5;  Gf → 6;  G → 7;  Gs → 8;  Gss → 9;
   Aff → 7;  Af → 8;  A → 9;  As → 10; Ass → 11;
   Bff → 9;  Bf → 10; B → 11; Bs → 12; Bss → 13
```

As you can see, a **case** expression allows multiple pattern matches on an expression without using equations. Note that layout applies to the body of a case expression and can be overriden as before, using a semicolon. (As in a function type signature, the right-pointing arrow in a **case** expression must be typed as "- >" on your computer keyboard.)

The body of a **case** expression observes layout just as a **let** expression, including that semicolons can be used, as above, to place more than one pattern match on the same line.

Converting an absolute pitch to a pitch is a bit more tricky because of enharmonic equivalences. For example, the absolute pitch 15 might correspond to either (*Ds*, 1) or (*Ef*, 1). Euterpea takes the approach of always returning a sharp in such ambiguous cases:

```
pitch    :: AbsPitch → Pitch
pitch ap =
   let (oct, n) = divMod ap 12
   in ([C, Cs, D, Ds, E, F, Fs, G, Gs, A, As, B] !! n, oct − 1)
```

Details: (!!) is Haskell's zero-based list-indexing function; *list* !! *n* returns the $(n + 1)$th element in *list*. *divMod x n* returns a pair (q, r), where q is the integer quotient of x divided by n, and r is the value of x modulo n.

Copyrighted material

Given *pitch* and *absPitch*, it is now easy to define a function *trans* that transposes pitches:

$$
\begin{aligned}
&trans \quad :: Int \rightarrow Pitch \rightarrow Pitch \\
&trans\ i\ p = pitch\ (absPitch\ p + i)
\end{aligned}
$$

With this definition, all of the operators and functions introduced in the previous chapter have been covered.

Exercise 2.3 Show that $abspitch\ (pitch\ ap) = ap$ and, up to enharmonic equivalences, $pitch\ (abspitch\ p) = p$.

Exercise 2.4 Show that $trans\ i\ (trans\ j\ p) = trans\ (i + j)\ p$.

Exercise 2.5 *Transpose* is part of the *Control* data type, which in turn is part of the *Music* data type. Its use in transposing a *Music* value is thus a kind of "annotation" – it doesn't really change the *Music* value, it just annotates it as something that is transposed.

Define instead a recursive function *transM* :: *AbsPitch* → *Music Pitch* → *Music Pitch* that actually changes each note in a *Music Pitch* value by transposing it by the interval represented by the first argument.

Hint: To do this properly, you will have to pattern match against the *Music* value, something like this:

```
transM ap (Prim (Note d p)) = ...
transM ap (Prim (Rest d))   = ...
transM ap (m1 :+: m2)       = ...
transM ap (m1 :=: m2)       = ...
transM ap (Modify...)       = ...
```

Copyrighted material

3
Polymorphic and Higher-Order Functions

Several examples of polymorphic data types were introduced in the first couple of chapters. In this chapter the focus is on *polymorphic functions*, which are most commonly defined over polymorphic data types.

The already familiar *list* is the protoypical example of a polymorphic data type, and it will be studied in depth in this chapter. Although lists have no direct musical connection, they are perhaps the most commonly used data type in Haskell and have many applications in computer music programming. But in addition, the *Music* data type is polymorphic, and several new functions that operate on it polymorphically will also be defined.

(A more detailed discussion of predefined polymorphic functions that operate on lists can be found in Appendix A.)

This chapter also introduces *higher-order functions*, which take one or more functions as arguments or return a function as a result (functions can also be placed in data structures). Higher-order functions permit the elegant and concise expression of many musical concepts. Together with polymorphism, higher-order functions substantially increase the programmer's expressive power and ability to reuse code.

Both of these new ideas naturally follow the foundations that have already been established.

3.1 Polymorphic Types

In previous chapters, examples of lists containing several different kinds of elements – integers, characters, pitch classes, and so on – were introduced, and we can well imagine situations requiring lists of other element types. Sometimes, however, it is not necessary to be so particular about the type of the elements. For example, suppose we wish to define a function *length* that

Copyrighted material

determines the number of elements in a list. It does not really matter whether the list contains integers, pitch classes, or even other lists – we can imagine computing the length in exactly the same way in each case. The obvious definition is:

$$\begin{aligned} &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

This recursive definition is self-explanatory. Indeed, we can read the equations as saying: "The length of the empty list is 0, and the length of a list whose first element is x and remainder is xs is 1 plus the length of xs."

But what should the type of *length* be? Intuitively, we would like to say that, for *any* type a, the type of *length* is $[a] \to \mathit{Integer}$. In mathematics we might write this as:

$$\mathit{length} :: (\forall\ a)\ [a] \to \mathit{Integer}$$

But in Haskell this is written simply as:

$$\mathit{length} :: [a] \to \mathit{Integer}$$

In other words, the universal quantification of the type variable a is implicit.

Details: Generic names for types, such as a above, are called *type variables*, and are uncapitalized to distinguish them from concrete types such as *Integer*.

So *length* can be applied to a list containing elements of *any* type. For example:

$$\begin{aligned} &\mathit{length}\ [1,2,3] &&\Longrightarrow 3 \\ &\mathit{length}\ [C,D,\mathit{Ef}\,] \Longrightarrow 3 \\ &\mathit{length}\ [[1],[\,],[2,3,4]] \Longrightarrow 3 \end{aligned}$$

Note that the type of the argument to *length* in the last example is $[[\mathit{Integer}]]$; that is, a list of lists of integers.

Here are two other examples of polymorphic list functions, which happen to be predefined in Haskell:

$$\begin{aligned} &\mathit{head} &&:: [a] \to a \\ &\mathit{head}\ (x{:}\,\_) &&= x \end{aligned}$$

$$\begin{aligned} &\mathit{tail} &&:: [a] \to [a] \\ &\mathit{tail}\ (\_ : xs) &&= xs \end{aligned}$$

Copyrighted material

This apparent anomaly can be resolved by noting that *map*, like *length*, *head*, and *tail*, does not really care what its list element types are, *as long as its functional argument can be applied to them*. Indeed, *map* is *polymorphic*, and its most general type is:

$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

This can be read as "*map* is a function that takes a function from any type *a* to any type *b*, and a list of *a*'s, and returns a list of *b*'s." The correspondence between the two *a*'s and between the two *b*'s is important: a function that converts *Int*'s to *Char*'s, for example, cannot be mapped over a list of *Char*'s. It is easy to see that in the case of *toAbsPitches*, *a* is instantiated as *Pitch* and *b* as *AbsPitch*, whereas in *toPitches*, *a* and *b* are instantiated as *AbsPitch* and *Pitch*, respectively.

Note, as we did in Section 2.2, that the above reasoning can be viewed as the abstraction principle at work at the type level.

Details: In Chapter 1 it was mentioned that every expression in Haskell has an associated type. But with polymorphism, we might wonder if there is just one type for every expression. For example, *map* could have any of these types:

$(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
$(Integer \rightarrow b) \rightarrow [Integer] \rightarrow [b]$
$(a \rightarrow Float) \rightarrow [a] \rightarrow [Float]$
$(Char \rightarrow Char) \rightarrow [Char] \rightarrow [Char]$

and so on, depending on how it will be used. However, notice that the first of these types is in some fundamental sense more general than the other three. In fact, every expression in Haskell has a unique type, known as its *principal type*, the least general type that captures all valid uses of the expression. The first type above is the principal type of *map*, since it captures all valid uses of *map* yet is less general than, for example, the type $a \rightarrow b \rightarrow c$. As another example, the principal type of *head* is $[a] \rightarrow a$; the types $[b] \rightarrow a$, $b \rightarrow a$, and even *a* are too general, whereas something like $[Integer] \rightarrow Integer$ is too specific. (The existence of unique principal types is the hallmark feature of the *Hindley-Milner type system* [5, 6] that forms the basis of the type systems of Haskell, ML [7], and several other functional languages [8].)

Copyrighted material