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.

1.1 The Note versus Signal Dichotomy

The field of computer music has grown astronomically over the past several decades, and the material can be structured and organized along several dimensions. A dimension that proves particularly useful with respect to a programming language is one that separates *high-level* musical concerns from *low-level* musical concerns. Since a “high-level” programming language – namely Haskell – is used to program at both of these musical levels, to avoid confusion, the terms *note level* and *signal level* will be used in the musical dimension.

At the *note level*, a *note* (i.e., pitch and duration) is the lowest musical entity that is considered, and everything else is built up from there. At this level, in addition to conventional representations of music, we can study interesting aspects of so-called algorithmic composition, including the use of fractals, grammar-based systems, stochastic processes, and so on. From this basis we can also study the harmonic and rhythmic *analysis* of music, although that is not currently an emphasis in this textbook. Haskell facilitates programming at this level through its powerful data abstraction facilities, higher-order functions, and declarative semantics.

In contrast, at the *signal level*, the focus is on the actual sound generated in a computer music application, and thus a *signal* is the lowest entity that is considered. Sound is concretely represented in a digital computer by a discrete sampling of the continuous audio signal at a high enough rate that human ears cannot distinguish the discrete from the continuous, usually 44,100 samples per second (the standard sampling rate used for CDs). But in Euterpea, these details are hidden: signals are treated abstractly as continuous quantities. This greatly eases the burden of programming with sequences of discrete values. At the signal level, we can study sound synthesis techniques (to simulate the sound of a conventional instrument, say, or something completely artificial), audio processing (e.g., determining the frequency spectrum of a signal), and special effects (reverb, panning, distortion, and so on).

Suppose that a musician is playing music using a metronome set at 96, which corresponds to 96 beats per minute. That means that one beat takes $^{60}/_{96} = 0.625$ seconds. At a stereo sampling rate of 44,100 samples per second, that in turn translates into $2 \times 0.625 \times 44{,}100 = 55{,}125$ samples, and each sample typically occupies several bytes[1] of computer memory.

[1] The storage size of a sample is called the *bit depth*. Modern audio hardware typically supports bit depths of 16 bits (2 bytes) to 32 bits (4 bytes).

In contrast, at the note level, we only need some kind of operator or data structure that says “play this note,” which requires a total of only a small handful of bytes. This dramatic difference highlights one of the key computational differences between programming at the note level and that at the signal level.

Of course, many computer music applications involve both the note level *and* the signal level, and indeed there needs to be a mechanism to mediate between the two. Although such mediation can take many forms, it is for the most part straightforward, which is another reason why the distinction between the note level and the signal level is so natural.

This textbook begins with exploration of the note level (Chapters 1–17) and follows with examination of the signal level (Chapters 18–23). If you are interested only in the signal level, you may wish to skip Chapters 9–17.

1.2 Basic Principles of Programming

Programming, in its broadest sense, is *problem solving*. It begins by recognizing problems that can and should be solved using a digital computer. Thus the first step in programming is answering the question “What problem am I trying to solve?”

Once the problem is understood, a solution must be found. This may not be easy, of course, and you may discover several solutions, so a way to measure success is needed. There are various dimensions in which to do this, including correctness (“Will I get the right answer?”) and efficiency (“Will it run fast enough, or use too much memory?”). But the distinction of which solution is better is not always clear, since the number of dimensions can be large, and programs will often excel in one dimension and do poorly in others. For example, there may be one solution that is fastest, one that uses the least amount of memory, and one that is easiest to understand. Deciding which to choose can be difficult, and is one of the more interesting challenges in programming.

The last measure of success mentioned earlier, clarity of a program, is somewhat elusive, being difficult to quantify and measure. Nevertheless, in large software systems, clarity is an especially important goal, since such systems are worked on by many people over long periods of time, and evolve considerably as they mature. Having easy-to-understand code makes it much easier to modify.

In the area of computer music, there is another reason why clarity is important: namely, that the code often represents the author’s thought process, musical intent, and artistic choices. A conventional musical score does not 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.

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

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:

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

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

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 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.

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.

Sometimes, however, an expression has a never-ending sequence of calculations. For example, if x is defined as:

$$x = x + 1$$

then here is what happens when trying to calculate the value of x:

$$\begin{aligned}
&x\\
&\Rightarrow x + 1\\
&\Rightarrow (x + 1) + 1\\
&\Rightarrow ((x + 1) + 1) + 1\\
&\Rightarrow (((x + 1) + 1) + 1) + 1\\
&\ldots
\end{aligned}$$

Similarly, if a function f is defined as:

$$f\ x = f\ (x - 1)$$

then an expression such as f 42 runs into a similar problem:

$$\begin{aligned}
&f\ 42\\
&\Rightarrow f\ 41\\
&\Rightarrow f\ 40\\
&\Rightarrow f\ 39\\
&\ldots
\end{aligned}$$

Both of these clearly result in a never-ending sequence of calculations. Such expressions are said to *diverge*, or not terminate. In such cases the symbol $\perp$, pronounced "bottom," is used to denote the value of the expression. This means that every diverging computation in Haskell denotes the same $\perp$ value,[4] reflecting the fact that, from an observer's point of view, there is nothing to distinguish one diverging computation from another.

1.5 Types

Every expression (and therefore every value) also has an associated *type*. It is helpful to think of types as sets of expressions (or values), in which members of the same set have much in common. Examples include the primitive atomic types *Integer* (the set of all integers) and *Char* (the set of all characters), the user-defined atomic type *PitchClass* (the set of all pitch classes, i.e., note names), as well as the structured types [*Integer*] and [*PitchClass*] (the sets of all lists of integers and lists of pitch classes, respectively), and *String* (the set of all Haskell strings).

[4] Technically, each type has its own version of $\perp$.

The association of an expression or value with its type is very useful, and there is a special way of expressing it in Haskell. Using the examples of values and types above:

```
D          :: PitchClass
42         :: Integer
'a'        :: Char
"Euterpea" :: String
[C, D, Ef] :: [PitchClass]
('b', 4)   :: (Char, Integer)
```

Each association of an expression with its type is called a *type signature*. Note the use of single quotes of the form `'x'` or `'x'` in these definitions to indicate single characters. It is important to recognize the single quote (or apostrophe) symbol as being distinct from the backquote symbol (typically found on the same key as ∼), which appears as ‘*x*‘ in this text and serves very specific syntactic purposes in Haskell, to be discussed later on.

Details: Note that the names of specific types are capitalized, such as *Integer* and *Char*, as are the names of some atomic values, such as *D* and *Fs*. These will never be confused in context, since things to the right of "::" are types, and things to the left are values. Note also that user-defined names of values are *not* capitalized, such as *simple* and *x*. This is not just a convention: it is required when programming in Haskell. In addition, the case of the other characters matters, too. For example, *test*, *teSt*, and *tEST* are all distinct names for values, as are *Test*, *TeST*, and *TEST* for types.

Details: Literal characters are written enclosed in single forward quotes (apostrophes), as in `'a'`, `'A'`, `'b'`, `','`, `'!'`, `' '` (a space), and so on. (There are some exceptions, however; see the Haskell Report for details.) Strings are written enclosed in double quote characters, as in `"Euterpea"` above. The connection between characters and strings will be explained in a later chapter.

The "::" should be read "has type," as in "42 has type *Integer*." Note that square braces are used both to construct a list value (the left-hand side of (::) above) and to describe its type (the right-hand side above). Analogously, the round braces used for pairs are used in the same way. But also note that all of the elements in a list, however long, must have the same type, whereas the elements of a pair can have different types.