# SCALES: Learning Multimedia in a Mixed-Paradigm Language

Cory D. Boatright
Computer Science Department
Hiram College
boatrightcd@hiram.edu

Brian T. Howard
Computer Science Department
DePauw University
bhoward@depauw.edu

## Abstract

Learning functional programming is difficult for beginning programmers, especially when the most common examples come from data structures such as lists and trees. We believe an environment that supports working with multimedia objects, such as graphics and music, will help a student learn functional programming in a more efficient and productive manner. For this reason, we have started development of a set of media libraries in the Scala programming language. Our plan is to create an integrated development environment built on Eclipse that provides access to these tools. In this paper, we explain the design of our libraries, which we have named the SCALES project, and show library functionality by providing examples of Sierpinski's Triangle, the Towers of Hanoi, and Frère Jacques.

## 1 Introduction

Over the years Computer Science departments everywhere have tried to find the optimal language and technique to teach programming to beginning majors. Furthermore, the different paradigms implemented by programming languages can make transitioning from one language to another very difficult. The functional paradigm is often particularly challenging to learn, and offerings to assist in this endeavor have been sparse. Many students must resign themselves to learning LISP, with emacs typically being their only IDE.

An advantage of teaching a functional language is that it can provide a clean environment in which to introduce recursion and abstraction. Features such as first-class function values and pattern-matching support powerful operations on recursive data, such as lists, while the absence of side-effecting assignment statements enables simple arguments about the results of execution. Such concepts make the effort of learning a functional language worth the rewards, but there is an obvious problem with the lack of support for functional languages.

We have developed graphics and MIDI libraries using the Scala programming language [11] to help bridge this gap. These libraries will be used in the creation of SCALES, a plugin for the Eclipse IDE [2]. We explain the capabilities of these libraries and provide examples of their use, illustrating SCALES' ability to solve some of the problems experienced while adapting to a new programming paradigm. Computer Science students often want to have some sort of visual or audible response in their programs, and the multimedia support in SCALES will provide such feedback. Additionally, the IDE described in more detail in Future Work will solve the problem of requiring the use of a text editor in the absence of a true IDE.

## 1.1 Scala

Scala is a functional and object oriented programming language that compiles to Java bytecode. It is being developed by the LAMP group at the École Polytechnique Fédérale de Lausanne. Often, a student is taught a language briefly in order to enforce specific concepts, and the language is then remembered only as a "nifty" footnote in their educational career. This early retirement of a language is normally due to poor support for the language or difficulty in distributing software to a range of users. Since Scala compiles to Java bytecode, a student appreciating Scala may continue to create and deploy software in this language without inconveniencing end users. Furthermore, since Scala's object model is compatible with Java's, the two languages may be easily intermixed in a single project; in particular, all of the Java standard library classes are directly usable within Scala.

The hybrid paradigm also makes programming in Scala a more natural transition in an introductory programming course. Scala bears a close resemblance to Java, and the learning curve between the two languages can be as simple as learning slightly different syntax. Many universities and colleges are teaching Java as a first language, and a short unit on functional programming in Scala would be smoother than a sudden transition to LISP. As an example, consider the code in Figure **??**, which show the recursive factorial function in Java, Scala, and LISP. Although they are clearly related, the Scala code lowers several of the barriers (*e.g.*, prefixed arithmetic operators, and the need to understand how the special form `if` differs from other function calls) presented by LISP.

```
int factorial(int n) {
 if (n <= 1) return 1;
 else return n * factorial(n - 1);
}

def factorial(n: Int): Int =
 if (n <= 1) 1
 else n * factorial(n - 1)

(defun factorial (n)
 (if (<= n 1)
  1
  (* n (factorial (- n 1)))))
```

Figure 1: Factorial in Java, Scala, and LISP

Despite Scala's similarity and compatibility with Java, the functional nature of Scala makes the languages substantially different. To see this in a more sophisticated example than factorial, consider the following code which implements the `while` statement as a function:

```
def While(test: => Boolean)
         (body: => Unit): Unit = {
 if (test) {
   body
   While(test)(body)
 }
}
```

This takes advantage of several functional features. The `While` function is written in "curried" form, where the result of applying `While` to the first argument is another function ready to be applied to the second argument. The `=>` markers on these arguments indicate that they are to be passed as "closures," wrapped up as anonymous functions which will be re-evaluated each time they are referenced. The second argument can be passed an entire block of statements; since all statements in Scala are expressions, this will be treated as an expression of type `Unit` (analogous to Java's `void` type). Finally, the `While` function is "tail-recursive"—the Scala compiler turns it into JVM code that branches from the bottom of the function back to the top, which avoids the stack overflow one would cause from writing similar code in Java. The result is that you can write

```
While (someTest) {
 someActions
}
```

and it will compile to code almost as efficient as the built-in `while` statement. Although this example itself might not be appropriate for an introductory programming class, the concepts demonstrated are very useful when implementing library classes such as in SCALES.

## 2 Two-Dimensional Graphics

Taking advantage of Scala's relationship with Java, we used the two-dimensional graphics provided in the standard Java API as a basis for our own libraries. Rather than merely creating wrapper classes to bridge the gap between the two languages, we saw fit to make some simplifications to the overall process of drawing the graphics. These improvements still allow for fast rendering of two-dimensional figures.

Our geometric shapes are first and foremost wrapper classes and are implemented as subclasses of their Java counterparts. To provide a nicer code interface, color and drawing are no longer handled explicitly by a graphics context. Rather, the classes in the library handle these aspects themselves, with the color property handled through method calls to the shape. We feel this is far more intuitive, as a student can create a red circle, rather than a circle that must be drawn with a graphics context specified to draw in red. A canvas class is provided to facilitate the drawing of the shapes, and basic shape primitives are provided for most simple geometries, as well as a generic polygon.

### 2.1 Sierpinski's Triangle

A main theme to functional programming is recursion. For this reason, the best example of recursion with two-dimensional graphics is inherently a fractal pattern. We have decided to use Sierpinski's Triangle as our pattern for the example, based on a Haskell example by Hudak [4]. Sierpinski's Triangle, also known as Sierpinski's Gasket, is a pattern formed from a triangle which is divided into three parts, and each part is in turn divided according to the same algorithm. The algorithm is run a set number of iterations or until a certain triangle size is attained, at which time the triangles are filled, presenting the pattern to the viewer.

To create Sierpinski's Triangle, we need to create only two methods. A third, not seen here, is currently used to display the image and is a supplement in place of the IDE, which will implement these libraries and handle the displaying of graphical results. The first method we need creates a filled right triangle. To create the triangle, we use a polygon from the SCALES library and define the points in the function. The end result is seen in Figure 1.

The second method needed for the frac-

```
def fillTri(x: Int, y: Int,
            size: Int): ShapeExt = {
 PolygonExt(List((x, y),
                 (x+size, y),
                 (x, y-size)),
            Color.BLUE, Color.BLACK)
}
```

Figure 2: Filled right triangle

tal construction is the recursive one. This method divides a triangle until a size limit is achieved. Three recursive calls are made with each pass of the method, one for each of the three parts of the triangle. The `+++` operator combines two shapes by drawing the first over the second; the result is another shape. By repeated use, it builds up the entire composite image to be displayed. The effect is similar to building a list recursively in LISP. Figure 2 contains this method. The picture generated by these two methods can be seen in Figure 3.

```
def sTri(x: Int, y: Int, size: Int,
         limit: Int): ShapeExt = {
 if (size <= limit) {
  fillTri(x, y, size)
 } else {
  val half = size / 2
  sTri(x, y, half, limit) +++
  sTri(x, y-half, half, limit) +++
  sTri(x+half, y, half, limit)
 }
}
```

Figure 3: Sierpinski's triangle

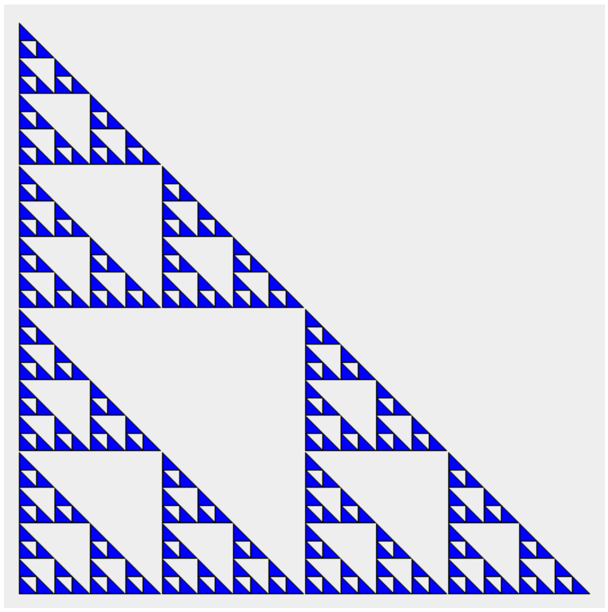
Figure 4: Sierpinski's triangle

## 3 Three-Dimensional Graphics

The original intent for three-dimensional graphics was to implement a library similar to the two-dimensional package. This proved to be impossible. Java has no three-dimensional graphics included in the standard API, so support for this type of multimedia had to be found elsewhere or created from more basic elements. We searched for several different existing three-dimensional libraries and found `JOGL` [10], `jMonkeyEngine` [8], and `Java3D` [6]. Due to the fact we had little background knowledge in graphics programming, we decided to implement our libraries with the assistance of `Java3D` because of the better tutorials and documentation.

`Java3D` uses the scene graph structure to generate a virtual world. In order to deal with the more complex workings of a scene graph, we could not merely create a set of Scala wrapper classes around Java objects. Rather, we created an abstraction for the user of SCALES by presenting geometric primitives as classes that can be operated on, rather than a node in a complex graph. The result of this development is a more intuitive package where a sphere can be rotated, rather than a sphere's geometric definition being the child of a rotation node in a tree.

Once a series of primitives were created, a further simplified world was built on top of the new SCALES three-dimensional package. This simple world consisted of a board with discrete coordinates, and methods that could be used on the board to add, move, and delete objects placed at each location. In addition, only the top object at each location can be operated on, and the board itself can be textured by an image created through use of the two-dimensional graphics package. For more advanced graphics work without discrete coordinates, the unrestricted three-dimensional package is still available for use. While more complex than the simplified world, the package still serves as an abstraction from the more complex techniques in graphics, such as lighting and matrix transformations.

## 3.1 Towers of Hanoi

Our second example is the classic "Towers of Hanoi." This algorithm is a favorite for showing recursion when teaching a functional programming language, but the output of the problem is generally given in verbal descriptions. While visual students may be able to see these descriptions in their head, actually viewing the solution in a three-dimensional world would be beneficial.

Using the package for a simplified three-dimensional world, we have created an example algorithm for the Towers of Hanoi. The solution given in code fragments 4, 5, and 6 works for any sized stack of disks, while the screenshot shown in Figure 7 shows the program while running with a tower size of 5. As with the two-dimensional example, code used to initialize the window has been left out.

```
val aBoard: Board = Board(3, 1)
aBoard.create(0, 0, 1.0, 0.5,
    Board.DISK, Color.LIGHT_GRAY)
aBoard.create(0, 0, 0.8, 0.5,
    Board.DISK, Color.RED)
aBoard.create(0, 0, 0.6, 0.5,
    Board.DISK, Color.BLUE)
aBoard.create(0, 0, 0.4, 0.5,
    Board.DISK, Color.WHITE)
aBoard.create(0, 0, 0.2, 0.5,
    Board.DISK, Color.YELLOW)
```

Figure 5: Creating Towers

```
def towers(height: Int, from: Int,
           to: Int, other: Int) = {
 if (height > 0) {
  towers(height-1, from, other, to)
  shift(from, to)
  towers(height-1, other, to, from)
 }
}
```

Figure 6: Solve the Towers

```
def shift(from: Int, to: Int) {
 Thread.sleep(1000)
 aBoard.move((from, 0), (to, 0))
}
```

Figure 7: Shift One Disk

In Figure 4, a board is first created that is three cells long and one cell wide. The number of cells determines the range of valid discrete coordinate locations, and all cells are the same size. Once the board is created, five disks are added to one space, and then the algorithm can be called to move them. When an object is added to the board the coordinates, relative size with regard to the cell, height, object type, and color must be provided.

When a shape is moved on the board, the origin and destination coordinates are provided as two tuples. If a shape is not at the top of a stack, it cannot be accessed at all when using this package. Not visible in the screenshot in Figure 7 is the fact that the camera is controllable via the keyboard. The single lighting source serves to show the three-dimensional nature of the objects and is static.

## 4 Music

In addition to graphics, we wanted to support generation of music. Unlike one current approach to using multimedia for introductory computer science [3], which works with sound at the level of individual amplitude samples, we decided to focus on a higher-level view of music as comprised of notes played by instruments. That is, we decided to work at the level of MIDI files instead of WAV files. This decision was influenced by the Haskore music library for Haskell [4] and also by the availability of two excellent Java libraries for working with musical notes and phrases, `jMusic` [9] and `JFugue` [7].

As with the graphics classes, we started by looking at writing Scala wrappers for one of these Java libraries. However, we found that the programming model of `jMusic` was too imperative, since it was based on the idea of creating phrase objects and then adding notes. For example, Figure 8 shows the `jMusic` code necessary to play a C major scale.

In contrast, the programming model of

```
val ph = new Phrase
ph.addNode(new Note(C5, q))
ph.addNode(new Note(D5, q))
ph.addNode(new Note(E5, q))
ph.addNode(new Note(F5, q))
ph.addNode(new Note(G5, q))
ph.addNode(new Note(A5, q))
ph.addNode(new Note(B5, q))
ph.addNode(new Note(C6, q))
```

Figure 9: Creating a Scale in `jMusic`

`JFugue` is optimized for creating simple melodies using the concept of a "music string." For example, the C major scale example in `JFugue` is simply `new Pattern("C D E F G A B C6")` (this takes advantage of `JFugue`'s defaults to octave five and quarter-note duration). While this is convenient for creating melodies, `JFugue` does not have an easy way to combine two melody lines in parallel within the same voice; it can handle simultaneous notes (chords, including certain cases where several short notes are played against a longer note), but in order to combine longer simultaneous phrases, the user must place them in separate voices.

Our choice was to implement a model based on Haskore. The primitive objects are individual notes, consisting of a pitch and a duration. These may be composed into phrases through either sequential composition, where each successive note starts when the preceding one ends, or through parallel composition, where all the notes start at the same time. Since each phrase itself has a duration, phrases may be composed with notes and other phrases in the same manner. To render a piece of music, the built-up structure of parallel and sequential phrases is flattened into a single list of notes with their start times, then handed to `jMusic` to be turned into a MIDI sequence.

```
object FrereJacques {
 def main(args: Array[String]) {
  val h = HALF_NOTE
  val q = QUARTER_NOTE
  val e = EIGHTH_NOTE

  val first  = N(F5, q) | N(G5, q) |
               N(A5, q) | N(F5, q)
  val second = N(A5, q) | N(BF5, q) |
               N(C6, h)
  val third  = N(C6, e) | N(D6, e) |
               N(C6, e) | N(BF5, e) |
               N(A5, q) | N(F5, q)
  val fourth = N(F5, q) | N(C5, q) |
               N(F5, h)
  val pause  = N(WHOLE_NOTE) * 2

  val melody = first*2 | second*2 |
               third*2 | fourth*2

  val score = new MyScore(100)
  score.add(melody &
            (pause | melody) &
            (pause * 2 | melody) &
            (pause * 3 | melody))
  Play.midi(score)
 }
}
```

Figure 10: Frère Jacques

### 4.1 Frère Jacques

The code in Figure 9 is a complete Scala program (minus some `import` statements) demonstrating the SCALES music classes. It plays a four-part round version of the traditional tune Frère Jacques (Figure 10). This demonstrates the higher-level, algebraic view of musical structure supported by our library. The tune is organized into four pairs of measures, each of which consists of a single four-beat measure played twice. The values `first`, `second`, `third`, and `fourth` contain the notes for these four distinct measures; the expression `first*2 | second*2` is the sequential composition of two copies of `first` followed by two copies of `second`. Finally, the expression `melody & (pause | melody)` creates the parallel composition of the first voice overlapping the second voice, which is delayed by a two-measure pause.

## 5 Future Work

Our long-term goals are to extend an existing Scala plugin for the Eclipse IDE in order to make multimedia output built into the system. With this implementation added, if a student's program uses the simple world, a graphics window will appear with the content displayed; the student will not need to worry about handling the graphical user interface code themselves. Similarly, if the music functions are used, a music player will automatically start appropriately. The existing IDE plugin's features will handle such things as syntax coloring and highlighting errors in the code prior to compilation.

In addition, we would like to add features both to the library and the IDE itself. Functional reactive programming [4] could be implemented into the two- or even three-dimensional libraries so that animation can also be achieved through functional methods; following [5], we hope to provide this in a common framework with the parallel and sequential composition of musical phrases. Similarly, lazy evaluation could be added as an example of another characteristic found in many functional programming languages. Lazy evaluation can be used in the displaying of fractal images that will retain their structure even when zoomed in. This would expand on our Sierpinski's Triangle example,

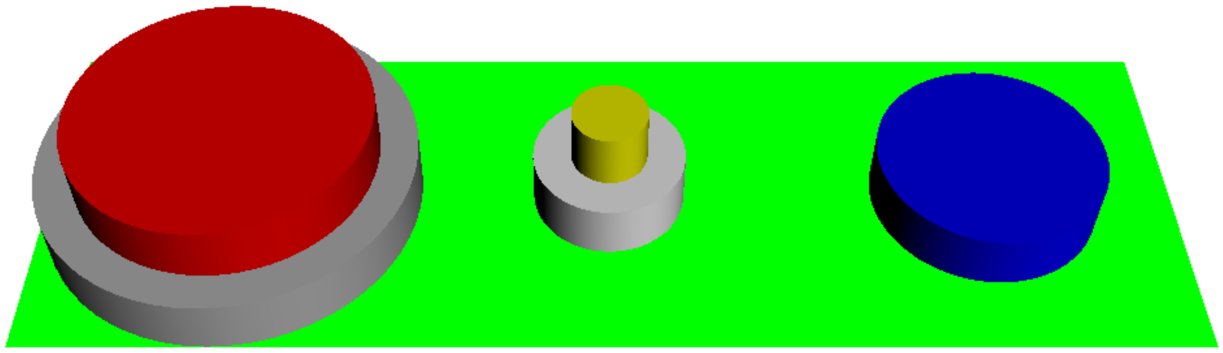

Figure 8: Towers of Hanoi

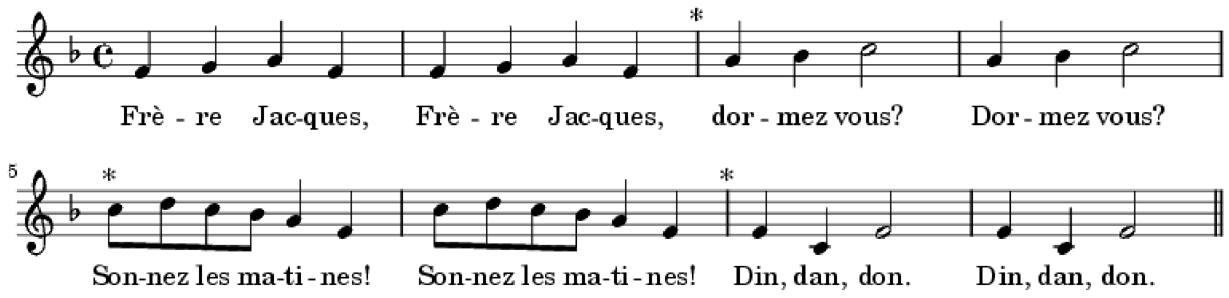
*
Frè - re Jac-ques, Frè - re Jac-ques, dor - mez vous? Dor - mez vous?
5
*
Son-nez les ma-ti - nes! Son-nez les ma-ti - nes! Din, dan, don. Din, dan, don.
*

Figure 11: Frère Jacques [12]

allowing it to be defined recursively past visible limits yet still allowing it to dynamically use only the iterations necessary for the current view.

A final feature we would like to implement into the SCALES project is language restriction. Scala is a powerful language with many characteristics from both an object-oriented and functional paradigm. When teaching the basics of functional programming, a professor may wish to have students activate a restricted subset to prevent students from drifting away from the functional features. This will serve to simplify the Scala language for the new student, a tactic also taken by DrScheme [1].

Our code is still in its first stages, so optimizations could be made. `Java3D` classes seem to provide an easier interface, but if performance becomes an issue, the three-dimensional library could be reimplemented using `JOGL`'s lower-level OpenGL API.

## 6 Acknowledgments

This work was supported by a National Science Foundation Research Experiences for Undergraduates grant, number IIS-0552370. Special thanks go to Dan Mateas, Simon's Rock College of Bard, who assisted in the implementation of some of the work reported here.

## References

[1] DrScheme. `http://www.drscheme.org/`.

[2] Eclipse. `http://www.eclipse.org/`.

[3] Mark Guzdial. *Introduction to Computing and Programming in Python: A Multimedia Approach.* Pearson Prentice Hall, 2005.

[4] Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000.

[5] Paul Hudak. An algebraic theory of polymorphic temporal media. Technical Report YALEU/DCS/RR-1259, Yale University, Department of Computer Science, July 2003.

[6] Java3D. `http://java3d.dev.java.net/`.

[7] JFugue. `http://www.jfugue.org/`.

[8] jMonkeyEngine. `http://www.jmonkeyengine.com/`.

[9] jMusic. `http://jmusic.ci.qut.edu.au/`.

[10] JOGL. `http://jogl.dev.java.net/`.

[11] Scala. `http://www.scala-lang.org/`.

[12] Wikipedia. Frère jacques — wikipedia, the free encyclopedia, 2007. [Online; accessed 29-September-2007].