GML: an extended Scala example

Mark Brown
mark@csse.unimelb.edu.au

October 2010

Contents

1 Introduction **5**

2 GML **5**

3 The virtual machine **6**
3.1 Locals and nonlocals 6
3.2 The machine state 8
3.3 Bytecode 10
3.4 Dynamic type checks 13
3.5 The main loop 14
3.6 Operators 15

4 Compiling **19**
4.1 Parsing 19
4.1.1 Lexical analysis 19
4.1.2 The parse tree 20
4.1.3 The parser 22
4.2 Program analysis 22
4.2.1 Nonlocal analysis 24
4.2.2 Liveness analysis 26
4.2.3 Local slot allocation 31
4.2.4 Code generation 36
4.3 Tying it together 39
4.3.1 The `Compiler` class 40
4.3.2 The `Main` object 41
4.3.3 Some GML Examples 43

5 Rendering **44**
5.1 Points and rays 44
5.2 Lights 46
5.3 Scene construction 49
5.4 Transformations 50

5.5 Boundaries and boundary lists . . . 56
5.6 The `World` class . . . 58
5.6.1 The base class and some subclasses . . . 59
5.6.2 The `Shape` subclass . . . 59
5.6.3 Quadratic roots . . . 61
5.6.4 Primitive shapes . . . 61
5.6.5 The `World` factory. . . . 68
5.7 Surface properties . . . 68
5.8 Dealing with surface acne . . . 69
5.9 Ray tracing . . . 71
5.10 Pixmaps . . . 74
5.11 The `render` operator . . . 75

6 Exercises for the reader **77**

A The full listing **79**
A.0.1 `LICENSE` . . . 79
A.1 The `compiler` package . . . 79
A.1.1 `compiler/CodeGen.scala` . . . 79
A.1.2 `compiler/Compiler.scala` . . . 81
A.1.3 `compiler/GmlLexer.scala` . . . 81
A.1.4 `compiler/GmlParser.scala` . . . 82
A.1.5 `compiler/Liveness.scala` . . . 83
A.1.6 `compiler/Location.scala` . . . 85
A.1.7 `compiler/Main.scala` . . . 85
A.1.8 `compiler/Nonlocals.scala` . . . 86
A.1.9 `compiler/SlotAlloc.scala` . . . 87
A.1.10 `compiler/Term.scala` . . . 89
A.2 The `runtime` package . . . 91
A.2.1 `runtime/Bytecode.scala` . . . 91
A.2.2 `runtime/DataStack.scala` . . . 92
A.2.3 `runtime/ExpandArray.scala` . . . 92
A.2.4 `runtime/Frames.scala` . . . 93
A.2.5 `runtime/Operators.scala` . . . 93
A.2.6 `runtime/Pixmap.scala` . . . 96
A.2.7 `runtime/Properties.scala` . . . 97
A.2.8 `runtime/Scene.scala` . . . 97
A.2.9 `runtime/Value.scala` . . . 98
A.2.10 `runtime/VM.scala` . . . 100
A.3 The `runtime.render` package . . . 101
A.3.1 `runtime/render/Boundary.scala` . . . 101
A.3.2 `runtime/render/Cone.scala` . . . 102
A.3.3 `runtime/render/Cube.scala` . . . 103
A.3.4 `runtime/render/Cylinder.scala` . . . 104
A.3.5 `runtime/render/Illumination.scala` . . . 104
A.3.6 `runtime/render/Light.scala` . . . 105

A.3.7 runtime/render/Plane.scala . . . 106
A.3.8 runtime/render/Point.scala . . . 107
A.3.9 runtime/render/Quadratic.scala . . . 107
A.3.10 runtime/render/Ray.scala . . . 108
A.3.11 runtime/render/Render.scala . . . 108
A.3.12 runtime/render/Shape.scala . . . 110
A.3.13 runtime/render/Sphere.scala . . . 111
A.3.14 runtime/render/Transform.scala . . . 112
A.3.15 runtime/render/View.scala . . . 113
A.3.16 runtime/render/World.scala . . . 114

List of Figures

1 Runtime values (incomplete). . . . 7
2 Support code for expanding arrays on demand. . . . 8
3 Virtual machine data stack and meta-stack. . . . 9
4 Virtual machine frame stack. . . . 10
5 The virtual machine. . . . 11
6 Bytecode definition. . . . 11
7 The Operators object. . . . 13
8 Base class for runtime values. . . . 14
9 Virtual machine support methods (runtime.VM). . . . 15
10 Virtual machine main loop (runtime.VM). . . . 16
11 Operator table support methods (runtime.Operators). . . . 17
12 Operator definitions (runtime.Operators). . . . 18
13 The GmlLexer class. . . . 19
14 Token parsers for names, graphic characters and string literals (compiler.GmlLexer). . . . 20
15 A token parser for numeric literals (compiler.GmlLexer). . . . 21
16 The parse tree (incomplete). . . . 21
17 The GmlParser class. . . . 23
18 The GML grammar (compiler.GmlParser). . . . 23
19 Parse tree fields for nonlocal analysis (compiler.Term). . . . 24
20 Nonlocal analysis. . . . 25
21 Parse tree fields for liveness analysis (compiler.Term). . . . 27
22 Liveness analysis. . . . 29
23 Liveness analysis of identifiers and binders (compiler.Liveness). 30
24 Liveness analysis of nested functions (compiler.Liveness). . . . 30
25 Liveness analysis of calls (compiler.Liveness). . . . 31
26 Parse tree fields for slot allocation (compiler.Term). . . . 32
27 Locations where bindings are stored. . . . 33
28 Slot allocation. . . . 34
29 Support methods for slot allocation (compiler.SlotAlloc). . . . 35
30 Slot allocation for nested functions (compiler.SlotAlloc). . . . 35
31 Slot allocation for identifiers and binders (compiler.SlotAlloc). 36

32 Slot allocation for calls (`compiler.SlotAlloc`). 37
33 Parse tree fields for code generation (`compiler.Term`). 37
34 Code generation. 38
35 Code generation of nested functions (`compiler.CodeGen`). 39
36 Code generation of array constructors, identifiers and binders (`compiler.CodeGen`). 40
37 Code generation of calls (`compiler.CodeGen`). 40
38 The `Compiler` class. 41
39 The `Main` object. 42
40 The `Point` class. 45
41 The `Ray` class. 46
42 The `Light` base class. 47
43 The `PlainLight` and `PointLight` classes. 47
44 The `SpotLight` class. 48
45 The `Scene` type. 49
46 Values for scene construction (`runtime.Value`). 49
47 Dynamic type checks for scene construction (`runtime.Value`). . 50
48 Lifting methods for scene construction (`runtime.Operators`). . . 51
49 Operators for scene construction (`runtime.Operators`). 52
50 The `Transform` class. 53
51 The `Transform` companion object. 54
52 The `View` class and companion object. 55
53 The `Boundary` class and companion object. 57
54 The `World` base case and some subclasses. 59
55 The `Shape` class and companion object. 60
56 Support methods for shapes (`runtime.render.Shape`). 62
57 The `Quadratic` object. 63
58 The `Sphere` class. 64
59 The `Cube` class. 65
60 The `Cylinder` class. 66
61 The `Cone` class. 67
62 The `Plane` class. 68
63 The `World` companion object. 69
64 The `Properties` class and companion object. 70
65 The `firstEntry` method (`runtime.render.Boundary`). 71
66 The `Render` class. 71
67 The `Illumination` class. 73
68 The `fireRay` method (`runtime.render.Render`). 74
69 The `Pixmap` class. 75
70 The rendering loop (`runtime.render.Render`). 76
71 The render operator (`runtime.VM`). 76

1 Introduction

This document contains an extended example of programming in Scala. The program, of which the full listing is supplied, implements a simple language for ray tracing named GML. Users write GML code that produces a scene description, which is then rendered to an output file. The task originally appeared as a challenge for the Third Annual ICFP Programming Contest in August, 2000.[1]

There are three main parts to our program. First is a virtual machine that provides an execution mechanism for the functional language. We describe the virtual machine in section 3. The second part is a compiler that reads in GML programs and produces bytecode suitable for the virtual machine. We describe the compiler in section 4. The last part is a renderer that performs ray tracing on a scene and produces the output. This is described in section 5.

Our design is by no means the best possible. Some potential improvements are collected together in section 6, in the form of exercises for the reader.

This extended example is aimed at "advanced" readers, in the sense that we don't take the time to introduce all the Scala language and library features we use. Instead, we try to focus our attention on unusual or tricky situations that can sometimes arise. We assume the reader has a reasonable understanding of Scala's main features and is able to consult other documentation if needed.

Why choose Scala for this task? The task covers a number of complex and disparate problems, ranging from program analysis to Boolean algebra and affine geometry. A high-level language is well suited to these kinds of problems. In particular, we'll make use of the "functional" style to simplify much of the reasoning we need to do about our program. Nevertheless, where they are of particular use we shall still use non-functional language features such as mutable stacks and maps. Scala is a good choice for this task because it provides ready access to the functional style, without requiring this style to be used throughout the program.

But why choose this task at all? Partly because finding solutions to these kinds of problems is interesting; mostly because the pictures we can produce are often astounding, and sometimes beautiful!

2 GML

This is a quick introduction to GML. The complete definition is in section 2 of the task specification, which can be found on the contest website. If you already know GML you can skip this part.

[1] `http://www.cs.cornell.edu/icfp/`

A GML program is a list of terms, where each term is either a function, an array or a token. A function is a list of terms surrounded by curly braces, and an array is a list of terms surrounded by square brackets. A token is an identifier, a binder (an identifier preceded by '`/`'), an operator or a literal value (a Boolean, integer, real or string).

Evaluation proceeds by stepping through each term in the list, maintaining an *environment* mapping identifiers to values and a *data stack* for intermediate results. An identifier is evaluated by pushing the value it maps to in the environment onto the data stack. A binder is evaluated by popping a value off the data stack and mapping the identifier to that value. Literals are pushed directly onto the data stack. The language defines a fixed set of operators that evaluate by popping inputs then pushing the result. To evaluate an array, we first evaluate its list of terms starting with a new data stack. The final contents of the data stack are converted into an array value, which is then pushed onto the original stack.

To evaluate a function we just push a value representing the function onto the stack. We don't evaluate the terms inside the function until later when we reach one of two important operators, `apply` and `if`. The `apply` operator saves the current environment, then pops a function value off the stack and evaluates it. When the evaluation finishes, the saved environment is restored. The `if` operator is similar except that it pops three values: a Boolean and two functions. If the Boolean is `true` it evaluates the first function, otherwise it evaluates the second.

Scene construction and rendering, which is the main purpose of a GML program, is all done via built-in operators. Since the implementation of these operators is somewhat orthogonal to the virtual machine and compiler, we leave further discussion of them to section 5.

3 The virtual machine

3.1 Locals and nonlocals

Consider the GML expression `{/x x y}`. When this function is pushed onto the data stack we will not know the value of `x`. That value will only become known after we apply the function, at which point `x` will become bound to the next value on the stack. In contrast, we will know the value of `y` because GML is lexically scoped; `y` must already be bound in the environment, and it keeps this value when the function is applied. This means that the runtime representation of the function must include a value for `y` but not for `x`.

We refer to identifiers like `x` as *locals*, and identifiers like `y` as *nonlocals*. An identifier occurrence in a function is local if there is a corresponding binder somewhere to its left, within the same or an enclosing scope. By "scope" we mean either the function itself or a nested array. For example, in `{/x [x]}` the identifier is local because the binder is in an enclosing scope, but in `{[/x] x}` it is nonlocal because the binder in the inner scope doesn't count.

```
package runtime

sealed abstract class Value

case class BoolValue(b: Boolean) extends Value {
  override def toString = b.toString
}

case class IntValue(n: Int) extends Value {
  override def toString = n.toString
}

case class RealValue(d: Double) extends Value {
  override def toString = d.toString
}

case class StringValue(s: String) extends Value {
  override def toString = "\""+ s +"\""
}

case class ArrayValue(elements: Array[Value]) extends Value {
  override def toString = "["+ elements.mkString(", ") +"]"
}

case class ClosureValue(address: Int, context: Array[Value]) extends Value {
  override def toString = "{&"+ address +", "+ context.mkString(", ") +"}"
}
```

Figure 1: Runtime values (incomplete).

The virtual machine uses *closures* to represent functions. A closure is a code address combined with an array of values, one for each of the function's nonlocals. We refer to this array as a *context*. Space will also be needed to store the values of locals when the function is applied. We shall refer to the space allocated for this purpose as a *frame*. Thus, the environment in our implementation is split into a context and a frame.

Figure 1 shows the runtime representation of values (more cases will be added later for scene descriptions). All of the cases here except `ClosureValue` are just wrappers around the corresponding Scala type. For closure values, the `address` parameter is an index into a bytecode array. The array will contain all of the bytecode for the entire program; the closure address value points to the start of the code for that closure. Our bytecode design is discussed in section 3.3.

We override to `toString` method so that values print in a way similar to GML terms. This can't completely be done however, because there is no GML equivalent of a closure address. We just print the address as an integer with a "`&`" before it.

```
package runtime

object ExpandArray {
  // Set the value at the index, resizing if needed.
  def set[T](array: Array[T], index: Int, value: T): Array[T] = {
    val newArray = if (index >= array.length) resize(array, index) else array
    newArray(index) = value
    newArray
  }

  // Double the array size until it includes the index.
  def resize[T](array: Array[T], index: Int): Array[T] = {
    var size = array.length
    if (size == 0) size = 1
    while (size <= index) size *= 2
    val newArray = new Array[T](size)
    array.copyToArray(newArray, 0)
    newArray
  }
}
```

Figure 2: Support code for expanding arrays on demand.

3.2 The machine state

The virtual machine requires three stacks: the data stack mentioned previously, a meta-stack and a frame stack. Along with the machine registers described below, these stacks comprise the virtual machine state.

The meta-stack is conceptually a stack of data stacks. It is used to support array construction. At the start of an array we push the current data stack onto the meta-stack. At the end of the array we make the current data stack into an array value then pop the previous data stack off the meta-stack.

The frame stack is used to support function calls. If a function has any locals, it allocates space for them by pushing a new frame onto the frame stack. Any function that does this is required to pop the frame before it returns, so that the caller regains access to its own frame. We refer to this as entering and leaving the frame, respectively.

But what happened to the context? In section 3.1 we said that the environment consists of a frame *and* a context. When calling a function we need to save the environment, which means saving the context as well as the frame. We do this by storing a pointer to the context in one of the frame slots; in later sections we'll show how this is achieved.

Figure 2 shows some code that will help us implement the virtual machine stacks. The `ExpandArray` object provides a method to set an array element, using an immutable style interface where the result array is returned. In most cases `set` behaves just like a mutable update and returns the same array that is passed to it without copying. However, if the index is out of range then a larger array is returned.

The data stack is implemented in Figure 3. It works in an obvious way,

```
package runtime

class DataStack {
  // Data stack.
  var stack = new Array[Value](256)
  var stackTop = 0
  var stackBottom = 0

  def push(value: Value): Unit = {
    stack = ExpandArray.set(stack, stackTop, value)
    stackTop += 1
  }

  def pop: Value = {
    if (stackTop == stackBottom) throw new Exception("gml: stack underflow")
    stackTop -= 1
    stack(stackTop)
  }

  // Meta-stack.
  var metaStack = new Array[Int](8)
  var metaStackTop = 0

  def startArray: Unit = {
    metaStack = ExpandArray.set(metaStack, metaStackTop, stackBottom)
    metaStackTop += 1
    stackBottom = stackTop
  }

  def endArray: Unit = {
    val elements = stack.slice(stackBottom, stackTop)
    stackTop = stackBottom
    if (metaStackTop == 0) throw new Exception("gml: meta stack underflow")
    metaStackTop -= 1
    stackBottom = metaStack(metaStackTop)
    push(ArrayValue(elements))
  }
}
```

Figure 3: Virtual machine data stack and meta-stack.

except that we have a separate stack pointer for the bottom of the stack. We shall use the extra pointer to implement the meta-stack.

The meta-stack, implemented as part of the same class, is actually a stack of `stackBottom` values. To start an array the current value of `stackBottom` is pushed, and it is then set to `stackTop`. Since `stackBottom` now equals `stackTop`, the data stack is effectively empty. At the end of the array constructor, we remove the stack entries between `stackBottom` and `stackTop` and copy them to the new array. We then pop the value of `stackBottom` that was earlier pushed, and push the new array onto the data stack as an `ArrayValue`.

Figure 4 shows the frame stack implementation. The frame pointer points to slot zero of the topmost frame, or is -1 if there are no frames. To access a frame

```
package runtime

class Frames {
  var frames = new Array[Value](4096)
  var framePtr = -1

  def enter(size: Int): Unit = {
    framePtr += size
    if (framePtr >= frames.length) frames = ExpandArray.resize(frames, framePtr)
  }

  def leave(size: Int): Unit = {
    framePtr -= size
    if (framePtr < -1) throw new Exception("gml: frame stack underflow")
  }

  def apply(slot: Int): Value =
    frames(framePtr - slot)

  def update(slot: Int, value: Value): Unit =
    frames(framePtr - slot) = value
}
```

Figure 4: Virtual machine frame stack.

slot, either when getting or setting, we simply subtract the slot number from the frame pointer. We use `apply` and `update` as the access method names, which in Scala allows the use of syntax which is similar to array access. For example, we get a slot value with `frames(slot)` and we set it with `frames(slot) = value`.

Aside from the stacks, the virtual machine state includes three registers. The `currentContext` register points to the context of the closure that is currently executing. The `codePointer` register contains the address of the next bytecode to be executed, and `returnPointer` contains the address we should return to when the current closure finishes. A negative value of `codePointer` indicates that the program should halt (so a negative value of `returnPointer` means we should halt when the current closure finishes). The `VM` class, which contains the stacks and registers and will later implement the main loop, is shown in Figure 5.

The paramater to the `VM` class is a bytecode array. Bytecode is described in the next section.

3.3 Bytecode

Bytecode is a way of representing virtual machine instructions using fixed-size chunks of data. This section describes the bytecode we use in our program. Note that we don't actually represent these in byte form, although it wouldn't be hard to encode them in two words (or even one) by using external tables for the operators and real and string literals.

Figure 6 shows the bytecode definition. We use case classes so we can pat-

```
package runtime

class VM(code: Array[Bytecode]) {
  // Stacks.
  val data = new DataStack
  val frames = new Frames

  // Registers.
  var currentContext: Array[Value] = null
  var codePointer = -1
  var returnPointer = -1

  ...
}
```

Figure 5: The virtual machine.

```
package runtime

sealed abstract class Bytecode

case object DoRender extends Bytecode
case class Apply(isTailCall: Boolean) extends Bytecode
case class If(isTailCall: Boolean) extends Bytecode
case object Return extends Bytecode
case class EnterFrame(size: Int) extends Bytecode
case class LeaveFrame(size: Int) extends Bytecode
case class SaveContext(slot: Int) extends Bytecode
case class RestoreContext(slot: Int) extends Bytecode
case class SaveReturnPtr(slot: Int) extends Bytecode
case class RestoreReturnPtr(slot: Int) extends Bytecode
case class MakeClosure(size: Int) extends Bytecode
case class PushInt(n: Int) extends Bytecode
case class PushReal(d: Double) extends Bytecode
case class PushString(s: String) extends Bytecode
case class PushNonlocal(slot: Int) extends Bytecode
case class PushLocal(slot: Int) extends Bytecode
case class PopLocal(slot: Int) extends Bytecode
case object PopUnused extends Bytecode
case class Op(op: Operators.OpType) extends Bytecode
case object StartArray extends Bytecode
case object EndArray extends Bytecode
```

Figure 6: Bytecode definition.

tern match on bytecode values, which will be useful when implementing the main loop of the virtual machine in section 3.5. Note that the cases with no parameters are actually case objects, not case classes. This makes sense because there is only ever one value in each of these cases—there are no parameters to distinguish between different occurrences. For example, every occurrence of `Return` is identical so they are all really the same value.

The first bytecode, `DoRender`, implements the GML `render` operator.

The next two cases, `Apply` and `If`, implement the `apply` and `if` operators that call a closure. To perform the call, we set `currentContext` to the closure's context, set `returnPointer` to the current value of `codePointer`, and set `codePointer` to the address of the closure. The `isTailCall` parameter is `true` if this is a tail call, which is a call occurring as the last term in a function. Tail calls are implemented in the same way as regular calls, except that `returnPointer` is left at its current value. This has the same effect as returning again immediately after the call, except that there is no need to save and restore `returnPointer`.

The next bytecode, `Return`, causes the current closure to return. We simply set `codePointer` to `returnPointer`.

`EnterFrame` and `LeaveFrame` create or destroy a frame with the given size. If a function enters a frame it must leave that frame before it returns, and it must use the same size in both cases. No frame slots may be accessed unless a frame of a suitable size has been entered (and not left).

`SaveContext` and `RestoreContext` copy the `currentContext` register to and from a frame slot, given by the `slot` parameter. `SaveReturnPtr` and `RestoreReturnPtr` do the same for the `returnPointer` register. Making a call destroys the values of these registers; if the caller needs to refer to either register after the call then it must save and restore the register to retain the correct value.

`MakeClosure` constructs a new closure value. The `size` parameter is equal to the size of the context. The virtual machine pops this many values to create the context, then pops the address value before pushing the new closure onto the stack.

`PushInt`, `PushReal` and `PushString` push literal values onto the stack. `PushNonlocal` and `PushLocal` push a value from the given slot of the context or frame, respectively.

`PopLocal` pops a value into the given frame slot. There is no corresponding bytecode to pop a nonlocal value; this wouldn't make sense, because the nonlocals are fixed when the context is created. `PopUnused` pops a value and discards it.

`Op` performs the action of an operator. The `op` parameter is a function acting on a data stack, which has the side effect of performing the operation. This type is defined in the `Operators` object, shown in Figure 7. The operator table will be defined in the `Operators.opTable` value, using a map literal. We provide `apply` and `isDefinedAt` methods so that `Operators` can be used like a partial function. We shall fill in the details in section 3.6.

Note that we don't count `render`, `apply` and `if` as operators since they each have their own specialized bytecode. Conversely, we count `true` and `false` as operators to save us the trouble of dealing with Booleans separately.

The last two bytecode cases, `StartArray` and `EndArray`, implement the start and end of array construction.

One observation about our bytecode design is worth making. The simplest possible function in GML is the identity, `{}`, which returns as soon as it is called, and the next simplest functions are ones that return a constant, such as `{ 1 }` or

```
package runtime

import Math._

object Operators {
  type OpType = DataStack => Unit

  def apply(name: String): OpType =
    opTable(name)

  def isDefinedAt(name: String): Boolean =
    opTable contains name

  val opTable: Map[String, OpType] = Map(
    ...
  )
}
```

Figure 7: The `Operators` object.

`{ x }`. In our bytecode, the identity is implemented as a single `Return` bytecode, and constant functions are implemented as a single `Push` (of the required type) followed by a `Return`. With no more than two instructions, the implementation of these functions is fast. This is an example of optimizing the common case: given that these functions, particularly the constant functions, are very common in GML, this goes a long way towards making our program efficient.

3.4 Dynamic type checks

GML is a dynamically typed language, but in our code we will need to access runtime values statically. For example, to implement a call we will need to pop a closure value off the stack and access its address and context, but the value we get from the stack is just a `Value`, not a `ClosureValue`. If the user writes a program such as `{1 apply}` then the value we get won't even be a closure, it will be an integer. In this case we need to halt the program and report a type error.

We will need to implement dynamic type checks to detect these errors and produce appropriate error messages. A good place to implement them is in the base class for runtime values. The implementation of `Value` in Figure 8 contains methods for either extracting the information in a more usable form or throwing an exception. In the latter case we call `typeError`, which has the unusual result type of `Nothing`. Since there are no values with this type, this precludes the method from ever returning normally (which it doesn't do because it always throws an exception). The reason we give it this type is because, in each place it is used, a different and incompatible result type is expected. Literally, `Nothing` satisfies these expectations.

Notice how we avoided using the name `toString`? If we hadn't, Scala would complain because this overrides the `toString` method defined in class `Any`, and

```
sealed abstract class Value {
  def toBoolean: Boolean = this match {
    case BoolValue(b) => b
    case _ => typeError("Boolean")
  }

  def toInt: Int = this match {
    case IntValue(n) => n
    case _ => typeError("Int")
  }

  def toReal: Double = this match {
    case RealValue(d) => d
    case _ => typeError("Real")
  }

  def toStr: String = this match {
    case StringValue(s) => s
    case _ => typeError("String")
  }

  def toArray: Array[Value] = this match {
    case ArrayValue(elements) => elements
    case _ => typeError("Array")
  }

  def toClosure: ClosureValue = this match {
    case c @ ClosureValue(_, _) => c
    case _ => typeError("Function")
  }

  private def typeError(s: String): Nothing =
    throw new Exception("type error: expected "+ s +" found "+ this)
}
```

Figure 8: Base class for runtime values.

we haven't used an `override` modifier. This is a good thing because if we did accidentally override that method the result would be disastrous. It would be called any time we tried to print out a `Value` or append it to a string, and in most cases, this would just call `typeError`... which would try to append the value to a string for the error message. The end result would be a stack overflow.

3.5 The main loop

Some support methods that use the dynamic type checks are shown in Figure 9. These will be called from the main loop to implement the construction and calling of closures. Note that, because the data is coming off a stack, we get the last pieces of data before earlier ones. So `callIf` pops the "else" value first and the condition last, which is the opposite order to how they are written in the GML program. Likewise, the `for` loop in `makeClosure` fills in the context

```
private def callClosure(value: Value, isTailCall: Boolean): Unit = {
  val closure = value.toClosure
  if (!isTailCall) returnPointer = codePointer
  codePointer = closure.address
  currentContext = closure.context
}

private def callIf(isTailCall: Boolean): Unit = {
  val elseVal = data.pop
  val thenVal = data.pop
  val condition = data.pop.toBoolean
  callClosure(if (condition) thenVal else elseVal, isTailCall)
}

private def makeClosure(size: Int): Unit = {
  val context = new Array[Value](size)
  for (i <- (0 until size).reverse) context(i) = data.pop
  data.push(ClosureValue(data.pop.toInt, context))
}
```

Figure 9: Virtual machine support methods (`runtime.VM`).

array in reverse.

The main loop of the virtual machine is shown in Figure 10. It first sets up the registers to the desired values, then evaluates bytecode until `codePointer` is negative. When it returns, it does not set the registers back to their previous values. If this is required, the caller will need to arrange it.

The `render` method will implement the GML rendering operator. We shall define this method in section 5.11.

The initial value of -1 for `returnPointer` is significant. As we said earlier, this will cause execution to halt at the end of the current closure. In particular, the while loop will continue running until we reach a `Return` bytecode. At that point the -1 value is copied to the `codePointer` register, causing the next loop test to fail and the loop to terminate.

The `currentContext` register is not initialized at all. It is assumed that the code being executed is at the top level where nonlocals are not allowed, so this register shouldn't be looked at until after a call. At this point, the register will be set to the context of the closure that is called.

3.6 Operators

The last thing required for our virtual machine is the operator table. We'll start with just the computational operators; the scene building operators will be left until section 5.

Figure 11 shows some support methods in the `Operators` object. These *lifting* methods will help write the operator definitions concisely. Each takes a function, which itself takes one or two arguments of type `Double` or `Int`, and carries this function out on a `DataStack`. That is, it pops the arguments, calls

```
def execute(address: Int): Unit = {
  codePointer = address
  returnPointer = -1
  while (codePointer >= 0) {
    val bytecode = code(codePointer)
    codePointer += 1
    bytecode match {
      case DoRender               => render
      case Apply(isTailCall)      => callClosure(data.pop, isTailCall)
      case If(isTailCall)         => callIf(isTailCall)
      case Return                 => codePointer = returnPointer
      case EnterFrame(size)       => frames.enter(size)
      case LeaveFrame(size)       => frames.leave(size)
      case SaveContext(slot)      => frames(slot) = ArrayValue(currentContext)
      case RestoreContext(slot)   => currentContext = frames(slot).toArray
      case SaveReturnPtr(slot)    => frames(slot) = IntValue(returnPointer)
      case RestoreReturnPtr(slot) => returnPointer = frames(slot).toInt
      case MakeClosure(size)      => makeClosure(size)
      case PushInt(n)             => data.push(IntValue(n))
      case PushReal(d)            => data.push(RealValue(d))
      case PushString(s)          => data.push(StringValue(s))
      case PushNonlocal(slot)     => data.push(currentContext(slot))
      case PushLocal(slot)        => data.push(frames(slot))
      case PopLocal(slot)         => frames(slot) = data.pop
      case PopUnused              => data.pop
      case Op(op)                 => op(data)
      case StartArray             => data.startArray
      case EndArray               => data.endArray
    }
  }
}

private def render: Unit = {
  ...
}
```

Figure 10: Virtual machine main loop (`runtime.VM`).

the function with these arguments, then pushes the result. The letters at the end of the method name indicate the number and type of arguments expected by the function. Note once again that, for the two-argument methods, the arguments are popped in reverse order.

We use the curried form of declaration for these methods because we want to supply the function argument in the operator table, but we won't supply the `DataStack` argument until the operator is used (this is the `op(data)` call in Figure 10).

The computational operators are defined in Figure 12. Using a map literal gives us a convenient way to write the table, and will also allow the parser to do fast lookups. The lifting methods certainly make defining operators much easier, but it's tempting to try to go further and declare them as `implicit`. Hopefully, we could then omit the methods and have Scala insert them where

```
def liftR(f: Double => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toReal))

def liftI(f: Int => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toInt))

def liftRR(f: (Double, Double) => Value)(data: DataStack) = {
  val arg2 = data.pop.toReal
  val arg1 = data.pop.toReal
  data.push(f(arg1, arg2))
}

def liftII(f: (Int, Int) => Value)(data: DataStack): Unit = {
  val arg2 = data.pop.toInt
  val arg1 = data.pop.toInt
  data.push(f(arg1, arg2))
}
```

Figure 11: Operator table support methods (`runtime.Operators`).

needed.

Unfortunately, this doesn't quite work with Scala's flow-based type inference algorithm.[2] The problem is that if an expression such as

```
x => RealValue(toDegrees(acos(x)))
```

appears where the type `DataStack => Unit` is expected, Scala not unreasonably guesses that `x` has type `DataStack` and `RealValue(toDegrees(acos(x)))` has type `Unit`. This causes two things to go wrong. First, if `x` has type `DataStack` then `acos(x)` is a type mismatch, because a `Double` is expected. Second, the body of the function literal *can* be treated as `Unit` by ignoring the result value. But this doesn't match the type of the implicit method, which expects an argument of type `Double => Value`, so the implicit method is not inserted and we get a type mismatch further up.

The general solution to these kinds of problems is to add type annotations. In this case both problems can be avoided if we add type annotations to the function literals. For example, the first three entries in the operator table could be written:

```
"acos" -> ((x => RealValue(toDegrees(acos(x)))): (Double => Value)),
"addf" -> (((x, y) => RealValue(x + y)): ((Double, Double) => Value)),
"addi" -> (((x, y) => IntValue(x + y)): ((Int, Int) => Value)),
```

This is rather more verbose than explicit lifting, so the cure is worse than the disease! Hence, we don't bother with the implicit lifting methods.

[2]This was tested with Scala 2.7.6.

```
val opTable: Map[String, OpType] = Map(
  "acos"       -> liftR(x => RealValue(toDegrees(acos(x)))),
  "addf"       -> liftRR((x, y) => RealValue(x + y)),
  "addi"       -> liftII((x, y) => IntValue(x + y)),
  "asin"       -> liftR(x => RealValue(toDegrees(asin(x)))),
  "clampf"     -> liftR(clamp),
  "cos"        -> liftR(x => RealValue(cos(toRadians(x)))),
  "divf"       -> liftRR((x, y) => RealValue(x / y)),
  "divi"       -> liftII((x, y) => IntValue(x / y)),
  "eqf"        -> liftRR((x, y) => BoolValue(x == y)),
  "eqi"        -> liftII((x, y) => BoolValue(x == y)),
  "false"      -> { _ push(BoolValue(false)) },
  "floor"      -> liftR(x => IntValue(floor(x).toInt)),
  "frac"       -> liftR(frac),
  "get"        -> doGet,
  "length"     -> doLength,
  "lessf"      -> liftRR((x, y) => BoolValue(x < y)),
  "lessi"      -> liftII((x, y) => BoolValue(x < y)),
  "modi"       -> liftII((x, y) => IntValue(x % y)),
  "mulf"       -> liftRR((x, y) => RealValue(x * y)),
  "muli"       -> liftII((x, y) => IntValue(x * y)),
  "negf"       -> liftR(x => RealValue(-x)),
  "negi"       -> liftI(x => IntValue(-x)),
  "real"       -> liftI(x => RealValue(x)),
  "sin"        -> liftR(x => RealValue(sin(toRadians(x)))),
  "sqrt"       -> liftR(x => RealValue(sqrt(x))),
  "subf"       -> liftRR((x, y) => RealValue(x - y)),
  "subi"       -> liftII((x, y) => IntValue(x - y)),
  "true"       -> { _ push(BoolValue(true)) }
)

private def clamp(d: Double): Value =
  RealValue(if (d < 0) 0 else if (d > 1) 1 else d)

private def frac(d: Double): Value =
  RealValue(d - (if (d < 0) ceil(d) else floor(d)))

private def doGet(data: DataStack): Unit = {
  val i = data.pop.toInt
  val a = data.pop.toArray
  data.push(a(i))
}

private def doLength(data: DataStack): Unit =
  data.push(IntValue(data.pop.toArray.length))
```

Figure 12: Operator definitions (`runtime.Operators`).

```
package compiler

import util.parsing.combinator.lexical.Lexical
import util.parsing.syntax.StdTokens

class GmlLexer extends Lexical with StdTokens {
  import util.parsing.input.CharArrayReader.EofCh
  def whitespace: Parser[Any] =
    rep(whitespaceChar | '%' ~ rep(chrExcept('\n', EofCh)))

  def token: Parser[Token] =
    ( name
    | graphic
    | numLiteral
    | stringLiteral
    )

  ...
}
```

Figure 13: The `GmlLexer` class.

4 Compiling

4.1 Parsing

We shall build a parser for GML using Scala's parsing combinator library, in package `scala.util.parsing.combinator`. To keep things simple, we can use the `StdTokens` trait defined in `scala.util.parsing.syntax`. Section 4.1.1 shows how we produce a sequence of GML tokens by extending `Lexical` with `StdTokens`. Section 4.1.2 describes the parse tree, and section 4.1.3 shows how it is constructed from a sequence of tokens by extending `StdTokenParsers`.

4.1.1 Lexical analysis

`StdTokens` in the Scala library defines four types of token: keywords, numeric literals, string literals, and identifiers. We'll use keyword tokens to represent graphic characters, and identifier tokens to represent GML operators and Boolean literals as well as GML identifiers (the parser will later sort out which of these are really identifiers, and which are operators). Numeric and string literal tokens will represent their GML counterparts.

The start of the `GmlLexer` class is shown in Figure 13. The `Lexical` superclass uses trait `Scanners`, which defines abstract methods `whitespace` and `token` that we need to implement. As the name suggests, `whitespace` defines a parser for anything that we wish to regard as whitespace. Aside from the usual whitespace characters we also want to treat comments as being equivalent to whitespace, where a comment consists of '`%`' followed by all characters up to the next newline or the end of the file (represented by `EofCh`).

The implementation of `token` is more involved, so we've broken it down

```
def name: Parser[Token] =
  letter ~ rep( letter | digit | '-' | '_' ) ^^ {
    case first ~ rest => Identifier(first::rest mkString "")
  }

def graphic: Parser[Token] =
  ( ('/': Parser[Char]) | '[' | ']' | '{' | '}' ) ^^
          { c => Keyword(c toString) }

def stringLiteral: Parser[Token] =
  ( '\"' ~> rep(chrExcept('\"', EofCh)) <~ '\"' ^^
          { cs => StringLit(cs mkString "") }
  | '\"' ~> failure("unclosed string literal")
  )
```

Figure 14: Token parsers for names, graphic characters and string literals (`compiler.GmlLexer`).

into smaller parts. Note that the parentheses are required around the different options. Without them, a semi-colon will be inferred after `name`, and Scala will then complain about the code on the next line.

Token parsers for names, graphic (punctuation) characters and string literals are shown in Figure 14. An important thing to notice in the graphic token parser is that we can't just write `('/' | ...)`, because Scala would convert the characters to `Int` for the '`|`' operation, and the resulting `Int` does not provide a method for `^^`. But we don't want the version of '`|`' that works on integers, we want the version that works on parsers! We can tell Scala this by using an type annotation: instead of `'/'` we write `('/':Parser[Char])`. Hence Scala knows that '`|`' is referring to the parser constructor, and the resulting parser can be used with `^^`.

String literals where the user has forgotten the closing quote would not be accepted by the first part of the `stringLiteral` rule. The second part of the rule is included to catch this case and fail, rather than letting the failure be reported from an unrelated (and probably baffling) place.

A numeric literal consists of an integer literal, an optional fraction part, and an optional exponent. The token parser is shown in Figure 15. Despite the fact that the literal is a number, we store the string representation in the token. The string is formed by concatenating one string for each part (the empty string is used for optional parts that are missing). In `optExp`, as with `graphic`, we use a type annotation so that Scala knows what type we want.

4.1.2 The parse tree

We shall start with a simple version of the parse tree, which is shown in Figure 16. This will be expanded in section 4.2.

Nodes in the parse tree are all members of the abstract class `Term`. Once again we use Scala's case classes, which will allow us to easily traverse the parse

```
  def numLiteral: Parser[Token] =
    intLiteral ~ optFrac ~ optExp ^^ {
      case i ~ f ~ e => NumericLit(i + f + e)
    }

  def intLiteral: Parser[String] =
    opt('-') ~ rep1(digit) ^^ {
      case None ~ digits => digits mkString ""
      case Some(_) ~ digits => '-'::digits mkString ""
    }

  def optFrac: Parser[String] =
    opt('.' ~> rep1(digit)) ^^ {
      case None => ""
      case Some(frac) => '.'::frac mkString ""
    }

  def optExp: Parser[String] =
    opt((('e': Parser[Char]) | 'E') ~> intLiteral) ^^ {
      case None => ""
      case Some(exp) => "e" + exp
    }
```

Figure 15: A token parser for numeric literals (`compiler.GmlLexer`).

```
package compiler

sealed abstract class Term

case class FunctionTerm(body: Array[Term]) extends Term
case class ArrayCtor(body: Array[Term]) extends Term
case class IntLiteral(n: Int) extends Term
case class RealLiteral(d: Double) extends Term
case class StringLiteral(s: String) extends Term
case class Operator(name: String) extends Term
case class Identifier(name: String) extends Term
case class Binder(name: String) extends Term
case object Render extends Term
case class Call(callType: CallType.Value) extends Term

object CallType extends Enumeration { val Apply, If = Value }
```

Figure 16: The parse tree (incomplete).

tree using pattern matching. There is one case for each type of GML term except Boolean literals—we shall treat Boolean literals as operators from now on. The cases have parameters representing either fixed pieces of data or the node children.

We make a separate case for `Operator` terms even though we store a name just like `Identifier` terms. The analysis passes need to know whether or not a name is an operator, but they don't need to know what the operation is. This means the parser needs to check whether the name exists in the operator table to know what type of term to build, but we don't need look up the implementation in the table until code generation (section 4.2.4).

The `Render` case is used for the GML "render" operator.

The `Call` case is used for the GML operators "apply" and "if"; as in section 3 these are not treated as regular operators. The reason we do this here is that these operators play a significant role in the analysis passes, so having a special case will simplify some of our later code.

The `CallType` object defines an enumeration distinguishing between the two types of call. In Scala, there is no language feature for enumerations. Instead, they are defined using the `Enumeration` class from the API, which provides a convenient and type safe way to define a set of constants.

4.1.3 The parser

Figure 17 shows the start of the `GmlParser` object. As with `GmlLexer`, there are two abstract members that need to be defined. In this case there is a type named `Tokens` and a value named `lexical`, from the `TokenParsers` supertrait of `StdTokenParsers`. The definitions need to correspond to our lexer, hence we use `StdTokens` as the type and we create a new `GmlLexer` for the lexical value.

The `parse` method parses an entire string and returns a `ParseResult`. If parsing is successful, the result contains the array of terms that were parsed. The result will also contain the remaining input, but since `phrase` only succeeds if it reads all input this part of the result is not interesting. If parsing is not successful, it is possible to extract the error message using the `msg` method.

To make the parser easier to use we also provide an `apply` method that parses a string and handles the result.

The rest of `GmlParser` is shown in Figure 18. The `processName` method first checks for the special cases of rendering and calls, then decides whether an identifier token is an operator or identifier. The `processNum` method decides whether a numeric literal is an integer or a real.

4.2 Program analysis

The program analysis passes traverse the parse tree and collect various kinds of information about nodes. We store this information inside the nodes; we will need to add some fields to the parse tree to accommodate this.

Each pass analyses one function term at a time (the entire program will itself be treated as a function body). The passes can assume that earlier passes

```
package compiler

import util.parsing.combinator.syntactical.StdTokenParsers
import util.parsing.syntax.StdTokens

object GmlParser extends StdTokenParsers {
  type Tokens = StdTokens
  val lexical = new GmlLexer

  def parse(s: String): ParseResult[Array[Term]] =
    phrase(terms)(new lexical.Scanner(s))

  def apply(s: String): Array[Term] =
    parse(s) match {
      case Success(terms, _) => terms
      case e: NoSuccess => throw new Exception("syntax error: "+ e.msg)
    }

  ...
}
```

Figure 17: The `GmlParser` class.

```
  def terms: Parser[Array[Term]] =
    rep(term) ^^ { Array.concat(_) }

  def term: Parser[Term] =
    ( ident                 ^^ { processName(_) }
    | "{" ~> terms <~ "}"   ^^ { FunctionTerm(_) }
    | "[" ~> terms <~ "]"   ^^ { ArrayCtor(_) }
    | "/" ~> ident          ^^ { Binder(_) }
    | numericLit            ^^ { processNum(_) }
    | stringLit             ^^ { StringLiteral(_) }
    )

  import runtime.Operators
  private def processName(name: String): Term =
    if (name == "render") Render
    else if (name == "apply") Call(CallType.Apply)
    else if (name == "if") Call(CallType.If)
    else if (Operators isDefinedAt name) Operator(name)
    else Identifier(name)

  private def processNum(num: String): Term =
    if (num matches "-?[0-9]+") IntLiteral(num.toInt)
    else RealLiteral(num.toDouble)
```

Figure 18: The GML grammar (`compiler.GmlParser`).

```
import collection.mutable.{HashMap, HashSet}

case class FunctionTerm(body: Array[Term]) extends Term {
  // Nonlocal slots.
  var nonlocals: Array[String] = null
  val nonlocalMap = new HashMap[String, Int]

  // Locals from the parent that are referenced here.
  val parentLocals = new HashSet[String]

  // Are nonlocals from the parent referenced here?
  var hasParentNonlocals = false
}

case class Identifier(name: String) extends Term {
  // Is the variable bound locally?
  var isLocal = false
}
```

Figure 19: Parse tree fields for nonlocal analysis (`compiler.Term`).

have all been performed on the function term, and that the current pass has been performed on all nested function terms. We will need to make sure these assumptions are justified when we implement the main loop of the compiler in section 4.3.1.

4.2.1 Nonlocal analysis

Nonlocal analysis determines which of the identifiers occurring in a function are nonlocal. It allocates a context slot for each nonlocal, and records which of them are local in the parent and whether any are nonlocal in the parent. Figure 19 shows the changes to the parse tree to add fields for the results.

The most convenient way to collect the required information is to traverse the function body from left to right. We keep track of the set of binders that we have seen, as well as the nonlocal slot allocations we have made so far. Figure 20 shows the `Nonlocals` class, which does the nonlocal analysis.

The `analyse` method traverses the function body. At the end, it copies the slot allocations, represented by a stack of names, to the function's `nonlocals` field. It also adds all of the nonlocals to the `parentLocals` field. This isn't correct yet: `parentLocals` should not contain *all* the nonlocals, only those that are local in the parent. When it comes time to process the parent we will need to fix up this field by removing the entries that aren't local. This is done in the code below.

Traversing the body involves pattern matching on each term in sequence. Functions and arrays are handled by separate methods. For identifiers, we record that there has been a reference to the name by calling `reference`. This returns `true` if the variable is local; we use this value to set the `isLocal` field of the identifier. For binders, we add the name to the set of locals. Other types

```
package compiler

import collection.mutable.{HashSet, Stack}

class Nonlocals(function: FunctionTerm) {
  // Binders seen.
  val locals = new HashSet[String]

  // Nonlocals allocated.
  val nonlocals = new Stack[String]

  // Do the analysis.
  def analyse: Unit = {
    traverse(function.body)
    function.nonlocals = nonlocals.toArray
    function.parentLocals ++= nonlocals
  }

  private def traverse(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => handleNested(nested)
      case ArrayCtor(body)            => handleArrayCtor(body)
      case id @ Identifier(name)      => id.isLocal = reference(name)
      case Binder(name)               => locals + name
      case _                          =>
    }

  private def handleNested(nested: FunctionTerm): Unit =
    for (name <- nested.nonlocals if !reference(name)) {
      nested.parentLocals - name
      nested.hasParentNonlocals = true
    }

  private def handleArrayCtor(body: Array[Term]): Unit = {
    val startLocals = locals.clone
    traverse(body)
    locals intersect startLocals
  }

  private def reference(name: String): Boolean = {
    val isLocal = locals contains name
    if (!isLocal && !(function.nonlocalMap contains name)) {
      function.nonlocalMap(name) = nonlocals.size
      nonlocals.push(name)
    }
    isLocal
  }
}
```

Figure 20: Nonlocal analysis.

of term are ignored.

To handle a nested function we need to reference all of its nonlocals. If any of them are nonlocal to the function we are analysing, this is where we need to fix up the `parentLocals` field. We remove the nonlocal and record that there was at least one nonlocal.

To handle an array constructor we need to recursively traverse its body. The only trick is that new binders can appear in the array, but these won't be in scope after the array has ended. We need to make sure the set of locals doesn't include any new binders from the array body. To do this, we clone the set of locals at the start of the array then intersect this cloned set at the end.

Finally, when an identifier is referenced we check if it is local. If it isn't, and we haven't seen it before, then we allocate a slot. We can determine the correct slot number by looking at the number of slots that have been allocated so far (that is, the size of the stack).

4.2.2 Liveness analysis

Consider the function term

```
{/x A /y B}
```

where `A` is a list of terms involving `x` and `B` is a list of terms involving `y` but not `x`. We will need to allocate frame slots for `x` and `y` to store the values that the binders receive. But since `x` does not appear in `B`, the same slot can be reused—the value stored there for `x` will no longer be needed by the time we need to store the value for `y`.

We shall say that `x` is *live* at a given point if there is an occurrence of `x` somewhere to the right in the same function term, and *dead* otherwise. For example, in the above function `x` is live in `A` (up to its last occurrence) but dead in `B`. The occurrence may be in a nested function or array, as long as it isn't "shadowed" by a binder for `x` that is in scope between the given point and `x`.

Liveness analysis calculates the point at which each identifier in a function becomes dead. This will allow us to optimize execution by determining the earliest time a frame slot can be reused or the entire frame freed. Although we reuse frame slots, it doesn't make sense to reuse context slots since these are fixed when the closure is created. But we can still determine when the last nonlocal becomes dead, and after this point the virtual machine will no longer need to keep track of the `currentContext` register.

The changes to the parse tree for this pass are shown in Figure 21. Identifiers can become dead either in an identifier term (if it is the last reference) or in a nested function, so we include information about the local(s) to free in these cases. We can leave the frame as soon as all of the locals are dead and no slots are being used to save registers across a call. This can happen at an identifier, a nested function or a call, so we add a flag for it to the base class. We also add a flag to binders to say whether the identifier is ever used; if it isn't, no frame slot will be needed at all.

```
sealed abstract class Term {
  // Should we leave the frame after this term?
  var leaveFrame = false
}

case class FunctionTerm(body: Array[Term]) extends Term {
  ...
  // Parent locals to free after the closure is built.
  var localsToFree: List[String] = Nil
}

case class Identifier(name: String) extends Term {
  ...
  // Should we free the local after pushing?
  var freeLocal = false
}

case class Binder(name: String) extends Term {
  // True if there are no references to the right.
  var isUnused = false
}

case class Call(callType: CallType.Value) extends Term {
  // Is this the last term in a function body?
  var isTailCall = false

  // Should we restore returnPointer after this call?
  var restoreReturnPointer = false

  // Should we save currentContext before this call?
  var saveCurrentContext = false

  // Should we restore currentContext after this call?
  var restoreCurrentContext = false

  // Should we free the currentContext slot after this call?
  var freeCurrentContext = false
}
```

Figure 21: Parse tree fields for liveness analysis (`compiler.Term`).

We detect various things about calls during liveness analysis. Tail calls are flagged. We flag calls before which the current context needs to be saved and after which it needs to be restored, and calls after which the return pointer needs to be restored. (The return pointer save will be added in the next pass.) We also flag the last call to restore the current context, which is where we can free the frame slot used for this purpose.

The start of the liveness analysis implementation is shown in Figure 22. For this pass the most convenient way to collect the required information is to traverse the body from right to left, which is why the body is reversed in the `for` expression. We keep track of a number of flags for things we have seen in our traversal, as well as the set of live locals. The set of live locals is the set of locals we have encountered for which we haven't yet encountered the binder.

The first action of liveness analysis is to check if the last term is a call. We use `lastOption` here because the function body may be empty. If it's not empty and the last term is a call, it is flagged as a tail call—tail calls will be ignored for the rest of this pass, so when we refer to "calls" below we will mean non-tail calls. After this check we do the traversal. We need to take action for nested functions, identifiers, binders and calls. Array constructors are recursively traversed, and all other terms are ignored.

Figure 23 shows the code to handle identifiers and binders. If an identifier is local we need to check if it is live. If it is not live, this means it is the last reference of that local and we should both free it and add it to the set of live locals. If we haven't seen any other locals this must be the last use of the frame, so we can leave the frame at this point. The `maybeLeaveFrame` method performs this check.

If an identifier is nonlocal we record that we have seen a nonlocal, and that it has been since the last call.

An identifier is dead immediately to the left of its binder, so when we encounter a binder we should ensure its identifier is removed from the live locals. If the identifier wasn't in the live locals, it means that the identifier wasn't used anywhere. In this case we flag the binder as unused.

Figure 24 shows the handling of nested functions, which is analogous to the handling of identifiers. Each of the nested `parentLocals` that is not currently live is added to `localsToFree` and `liveLocals`, and we check if we should leave the frame. If there are any nested nonlocals, we record that we have seen a nonlocal and that it has been since the last call.

Figure 25 shows the code to handle (non-tail) calls. If this is this last call, then this is the place we will need to restore the return pointer. Since this effectively frees the slot used by the return pointer, we check if we can leave the frame.

Any call which has nonlocal references after it needs to have the current context saved. We flag all calls satisfying this condition, although only the first call needs to actually save the register because we leave the value in place for later calls; we don't free the slot until after the last time it needs to be restored. In the next pass we will pick out the first call and reset the flag for all the others.

```
package compiler

import collection.mutable.HashSet

class Liveness(function: FunctionTerm) {
  // Have we seen a (non-tail) call?
  var seenCall = false

  // Have we seen a call with nonlocal references after it?
  var seenCallWithRestore = false

  // Have we seen a nonlocal reference?
  var seenNonlocal = false

  // Have we seen a nonlocal reference since the last call?
  var seenNonlocalSinceCall = false

  // Have we seen a local reference?
  var seenLocal = false

  // Currently live locals.
  val liveLocals = new HashSet[String]

  // Do the analysis.
  def analyse: Unit = {
    function.body.lastOption match {
      case Some(c @ Call(_)) => c.isTailCall = true
      case _ =>
    }
    traverse(function.body)
  }

  private def traverse(body: Array[Term]): Unit =
    for (term <- body.reverse) term match {
      case nested @ FunctionTerm(_)   => handleNested(nested)
      case ArrayCtor(arrayBody)       => traverse(arrayBody)
      case id @ Identifier(_)         => handleIdentifier(id)
      case b @ Binder(_)              => handleBinder(b)
      case c @ Call(_)                => handleCall(c)
      case _                          =>
    }

  ...
}
```

Figure 22: Liveness analysis.

```
private def handleIdentifier(id: Identifier): Unit =
  if (id.isLocal) {
    if (!(liveLocals contains id.name)) {
      // Last reference to this local.
      id.freeLocal = true
      liveLocals + id.name
      maybeLeaveFrame(id)
    }
  } else {
    seenNonlocal = true
    seenNonlocalSinceCall = true
  }

private def handleBinder(b: Binder): Unit =
  if (liveLocals contains b.name) liveLocals - b.name
  else b.isUnused = true

private def maybeLeaveFrame(term: Term): Unit =
  if (!seenLocal) {
    seenLocal = true
    term.leaveFrame = true
  }
```

Figure 23: Liveness analysis of identifiers and binders (`compiler.Liveness`).

```
private def handleNested(nested: FunctionTerm): Unit = {
  for (name <- nested.parentLocals if !(liveLocals contains name)) {
    // Closure construction is the last to use this local.
    nested.localsToFree = name :: nested.localsToFree
    liveLocals + name
    maybeLeaveFrame(nested)
  }

  if (nested.hasParentNonlocals) {
    seenNonlocal = true
    seenNonlocalSinceCall = true
  }
}
```

Figure 24: Liveness analysis of nested functions (`compiler.Liveness`).

```
private def handleCall(c: Call): Unit =
  if (!c.isTailCall) {
    // Restore returnPointer if this is the last call.
    if (!seenCall) {
      seenCall = true
      c.restoreReturnPointer = true
      maybeLeaveFrame(c)
    }

    // Save currentContext if there are nonlocal references after the call.
    if (seenNonlocal) c.saveCurrentContext = true

    // Restore currentContext if nonlocals are referenced before the next
    // call.  Free the currentContext slot on the last such occasion.
    if (seenNonlocalSinceCall) {
      seenNonlocalSinceCall = false
      c.restoreCurrentContext = true
      if (!seenCallWithRestore) {
        seenCallWithRestore = true
        c.freeCurrentContext = true
      }
    }
  }
```

Figure 25: Liveness analysis of calls (`compiler.Liveness`).

Restoring the current context needs to be done if any nonlocals are referenced before the next call. Nonlocals *after* the next call don't count, because in that case the restore will occur at the later call. In our backwards traversal we set the flag if we have seen any nonlocals since a call, and, since this term is itself a call, we record that we have not seen any nonlocals since a call. This is also where we check if the restore is the last one, in which case we set the flag for freeing the current context slot.

4.2.3 Local slot allocation

The slot allocation pass allocates a slot number for each of the locals. We traverse left to right over the function body, allocating slots when a binder is encountered and using the allocated slot when a local identifier is encountered. When the liveness information tells us that a slot can be freed, we add it to a "free list". Each time we allocate a slot we first check if there are any available on the free list. A new slot is only allocated if none are currently free.

Figure 26 shows the extra fields we need in the parse tree. In the base class we define `enterFrame`, which corresponds to the `leaveFrame` field from the previous pass. The remaining new fields store information about allocated slots.

The `Location` type is used where the location could either be in the frame or in the context. Its implementation is in Figure 27. Each case has a parameter for the slot number; the case itself says whether the slot is in the frame or in the

```
sealed abstract class Term {
  // Should we enter the frame before this term?
  var enterFrame = false

  ...
}

case class FunctionTerm(body: Array[Term]) extends Term {
  ...
  // Where to find nonlocal bindings in the parent.
  var locations: Array[Location] = null

  // Number of frame slots required.
  var frameSize = -1
}

case class Identifier(name: String) extends Term {
  ...
  // Where is the binding stored?
  var location: Location = null
}

case class Binder(name: String) extends Term {
  ...
  // Which frame slot is the value stored in?
  var frameSlot = -1
}

case class Call(callType: CallType.Value) extends Term {
  ...
  // Should we save returnPointer before this call?
  var saveReturnPointer = false

  // In which frame slots are the registers stored?
  var returnPointerSlot = -1
  var currentContextSlot = -1
}
```

Figure 26: Parse tree fields for slot allocation (`compiler.Term`).

```
package compiler

import runtime.{Bytecode, PushLocal, PushNonlocal}

sealed abstract class Location {
  def bytecode: Bytecode
}

case class FrameSlot(slot: Int) extends Location {
  def bytecode = PushLocal(slot)
}

case class ContextSlot(slot: Int) extends Location {
  def bytecode = PushNonlocal(slot)
}
```

Figure 27: Locations where bindings are stored.

context. When it comes to generating code, we will want to push the value at the location. The `bytecode` method returns the appropriate bytecode for this.

The start of the `SlotAlloc` class is shown in Figure 28. It maintains a map of the current slot allocations, the free list, and a number of flags.

We allocate slots for the virtual machine registers in the same way that we allocate them for locals. To do this we need a fixed name for each of the registers. The fields `nameReturnPointer` and `nameCurrentContext` provide these names. We have chosen names beginning with an underscore because these are not valid GML identifiers—the parser never produces binders or identifiers with these names, so there is no possibility of an accidental conflict with a user's GML program.

Slot allocation is performed by traversing over the function body in a similar way to previous passes. After the traversal is complete, we set the function frame size to the value of `numFrameSlots`. This value is equal to the maximum number of slots used at any point in the function. Ultimately, this will determine the size we use when entering and leaving the frame for this function.

Support methods for local slot allocation are shown in Figure 29. These include methods for handling the free list as described earlier, and a method `maybeEnterFrame` which corresponds to `maybeLeaveFrame` from the previous section.

The code for handling nested functions is shown in Figure 30. We need to set the `locations` field to an array containing a `Location` for each of the nonlocals in the function. When we emit bytecode to construct a closure for this function, the locations field will tell us where to find all the bindings for the context. The location depends on whether the identifier is local or nonlocal in the parent: local bindings are found in the frame slot that has been allocated, and nonlocal bindings are found in the context slot given by the parent function's `nonlocalMap`. We use a higher-order "map" expression (really just a method call to `map` with a function argument) to write this concisely.

```
package compiler

import collection.mutable.HashMap

class SlotAlloc(function: FunctionTerm) {
  // Current slot allocation.
  val frameSlots = new HashMap[String, Int]

  // Number of slots allocated.
  var numFrameSlots = 0

  // Frame slots no longer in use.
  var freeList: List[Int] = Nil

  // Have we entered the frame?
  var frameEntered = false

  // Have we saved returnPointer?
  var savedReturnPointer = false
  val nameReturnPointer = "_returnPointer"

  // Have we saved currentContext?
  var savedCurrentContext = false
  val nameCurrentContext = "_currentContext"

  // Allocate the slots.
  def allocate: Unit = {
    traverse(function.body)
    function.frameSize = numFrameSlots
  }

  private def traverse(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => handleNested(nested)
      case ArrayCtor(arrayBody)       => traverse(arrayBody)
      case id @ Identifier(_)         => handleIdentifier(id)
      case b @ Binder(_)              => handleBinder(b)
      case c @ Call(_)                => handleCall(c)
      case _                          =>
    }

  ...
}
```

Figure 28: Slot allocation.

```
private def maybeEnterFrame(term: Term): Unit =
  if (!frameEntered) {
    frameEntered = true
    term.enterFrame = true
  }

private def allocateSlot(name: String): Int = {
  var slot = -1
  freeList match {
    case firstFree :: rest =>
      slot = firstFree
      freeList = rest
    case Nil =>
      slot = numFrameSlots
      numFrameSlots += 1
  }
  frameSlots(name) = slot
  slot
}

private def freeSlot(name: String): Unit = {
  freeList = frameSlots(name) :: freeList
  frameSlots.removeKey(name)
}
```

Figure 29: Support methods for slot allocation (`compiler.SlotAlloc`).

```
private def handleNested(nested: FunctionTerm): Unit = {
  // Set the locations of the nested nonlocals.
  nested.locations = nested.nonlocals map { arg =>
    if (nested.parentLocals contains arg) FrameSlot(frameSlots(arg))
    else ContextSlot(function.nonlocalMap(arg))
  }

  // Free all parent locals whose last appearance is in the nested function.
  nested.localsToFree.foreach(freeSlot)
}
```

Figure 30: Slot allocation for nested functions (`compiler.SlotAlloc`).

```
private def handleIdentifier(id: Identifier): Unit =
  if (id.isLocal) {
    id.location = FrameSlot(frameSlots(id.name))
    if (id.freeLocal) freeSlot(id.name)
  } else {
    id.location = ContextSlot(function.nonlocalMap(id.name))
  }

private def handleBinder(b: Binder): Unit =
  if (!b.isUnused) {
    maybeEnterFrame(b)
    b.frameSlot = allocateSlot(b.name)
  }
```

Figure 31: Slot allocation for identifiers and binders (`compiler.SlotAlloc`).

The other task for handling nested functions is to free all of the locals in the `localsToFree` field. This needs to be done *after* creating the `locations` array because freeing a slot removes its entry from `frameSlots`, but that entry would have been needed to get the correct frame slot location.

Identifiers and binders are handled in Figure 31. Identifiers are handled similarly to nested functions: we need to set the location and then free the identifier if it is local. For a binder, if it is used at all then it needs a slot allocated. Since this may be the first time any frame slot is used, we need to check whether to enter the frame.

The last piece of the slot allocation pass is for handling calls, and is shown in Figure 32. As in the previous pass, we don't need to do anything for tail calls. For non-tail calls, we need to deal with the saving and restoring of the registers. This involves setting the flag and slot location in the `Call`, and possibly freeing the slot. Since this might be the first time a frame slot is used, we again need to check whether to enter the frame.

The return pointer must be saved before the first call, so we do this the first time then set `savedReturnPointer` to say that this has been done. The current context needs to be saved if the `saveCurrentContext` flag was set in the previous pass, but recall that we only wanted to do this for the first such call. Therefore, we set `savedCurrentContext` on the first occasion, and on later occasions we clear the flag in the `Call` rather than doing the save.

Restoring the registers is done based on the `restoreReturnPointer` and `restoreCurrentContext` flags. For the return pointer, there will only be one restore so we unconditionally free the slot. The current context may be restored multiple times, which is why we needed a separate `freeCurrentContext` flag. We only free the slot if this is set.

4.2.4 Code generation

Now that we have done the hard analysis work, code generation is relatively easy. It works in a similar way to the previous passes, in that we traverse one

```
  private def handleCall(c: Call): Unit =
    if (!c.isTailCall) {
      maybeEnterFrame(c)

      if (!savedReturnPointer) {
        savedReturnPointer = true
        c.saveReturnPointer = true
        c.returnPointerSlot = allocateSlot(nameReturnPointer)
      }

      if (c.saveCurrentContext) {
        if (!savedCurrentContext) {
          savedCurrentContext = true
          c.currentContextSlot = allocateSlot(nameCurrentContext)
        } else {
          c.saveCurrentContext = false
        }
      }

      if (c.restoreReturnPointer) {
        c.returnPointerSlot = frameSlots(nameReturnPointer)
        freeSlot(nameReturnPointer)
      }

      if (c.restoreCurrentContext)
        c.currentContextSlot = frameSlots(nameCurrentContext)

      if (c.freeCurrentContext)
        freeSlot(nameCurrentContext)
    }
```

Figure 32: Slot allocation for calls (`compiler.SlotAlloc`).

```
case class FunctionTerm(body: Array[Term]) extends Term {
  ...
  // Address of function in generated bytecode.
  var address = -1
}
```

Figure 33: Parse tree fields for code generation (`compiler.Term`).

function body at a time. In this case, however, we generate code for the function as a side effect.

Figure 33 shows the last addition we will need to make to the parse tree. This is the address of the start of the bytecode generated for the function, which we fill when we start to generate the code. As with the analysis passes, we generate code for all nested functions before we generate code for the parent, so the address will be valid in the nested functions by the time we need to create closures for them in the parent.

The start of the code generator is shown in Figure 34. The `code` parameter is a bytecode stack where we will generate code—an instruction is generated by

```
package compiler

import collection.mutable.Stack
import runtime._

class CodeGen(function: FunctionTerm, code: Stack[Bytecode]) {
  // Generate code for the function.
  def generate: Unit = {
    function.address = code.size
    traverse(function.body)
    function.body.lastOption match {
      case Some(c @ Call(_)) if (c.isTailCall) =>
      case _ => code push Return
    }
  }

  private def traverse(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => generateNested(nested)
      case ArrayCtor(arrayBody)       => generateArrayCtor(arrayBody)
      case id @ Identifier(_)         => generateIdentifier(id)
      case b @ Binder(_)              => generateBinder(b)
      case c @ Call(_)                => generateCall(c)
      case Render                     => code push DoRender
      case Operator(name)             => code push Op(Operators(name))
      case IntLiteral(n)              => code push PushInt(n)
      case RealLiteral(d)             => code push PushReal(d)
      case StringLiteral(s)           => code push PushString(s)
    }

  ...
}
```

Figure 34: Code generation.

pushing it onto this stack. The next address that will be used is given by the size of the code stack; we use this to set the function address at the start of `generate`.

After we have traversed the body we will need to generate a `Return` instruction unless the last term was a tail call. The `match` expression tests for this case first, in which case it generates nothing, and generates the instruction in all other cases (including when the function body is empty).

Traversal is from left to right. Literals and operators only require a single instruction to be generated; in the case of operators other than `render` we need to look up the operator implementation in the `Operators` object. The other cases are handled by separate methods.

Code generation of nested functions is shown in Figure 35. Since both the `enterFrame` and the `leaveFrame` flags could be set for this type of term, we check these at the start and end of the method, respectively. Each check needs to generate the corresponding bytecode if the flag is set. We will want to use the checks below, as well, so we define methods `maybeEnterFrame` and

```
private def generateNested(nested: FunctionTerm): Unit = {
  maybeEnterFrame(nested)
  code push PushInt(nested.address)
  nested.locations foreach { code push _.bytecode }
  code push MakeClosure(nested.locations.length)
  maybeLeaveFrame(nested)
}

private def maybeEnterFrame(term: Term): Unit =
  if (term.enterFrame) code push EnterFrame(function.frameSize)

private def maybeLeaveFrame(term: Term): Unit =
  if (term.leaveFrame) code push LeaveFrame(function.frameSize)
```

Figure 35: Code generation of nested functions (`compiler.CodeGen`).

`maybeLeaveFrame` to do the checks.

In between the calls to handle the frame, we generate code that constructs a closure for the nested function. We first generate code to push the bytecode address and all the arguments, whose locations can be found in the `locations` field of the nested function. For the arguments, we use the `foreach` method in a way similar to `map` from Figure 30. We then generate a `MakeClosure` instruction, passing it the number of arguments we have just pushed. At runtime, all of the arguments we pushed will be converted into a context, and this will be combined with the address to form the closure.

Figure 36 shows the code generation of array constructors, identifiers and binders. To construct an array we just generate the code for the array body, but surround it with `StartArray` and `EndArray` instructions. For identifiers, we generate the bytecode to push the value at its location onto the data stack. We also check whether to leave the frame afterwards. For binders, we first check whether to enter the frame. The bytecode for the binding depends on whether the binder is used or not. If it's unused we generate `PopUnused`, otherwise we generate `PopLocal`, with the correct frame slot for that binder.

Code generation of calls is shown in Figure 37. Prior to the call we check whether to enter the frame, and check whether to save each of the two virtual machine registers. The call itself is just the bytecode corresponding to the call type. After the call we check whether to restore the registers and whether to leave the frame.

4.3 Tying it together

In this section we implement a `Compiler` class that brings together the analysis passes, and a top level application for parsing and running GML programs. At this stage we will have a functioning system, although obviously it won't be possible to do any rendering until we implement the operators for it in section 5.

```
private def generateArrayCtor(body: Array[Term]): Unit = {
  code push StartArray
  traverse(body)
  code push EndArray
}

private def generateIdentifier(id: Identifier): Unit = {
  code push id.location.bytecode
  maybeLeaveFrame(id)
}

private def generateBinder(b: Binder): Unit = {
  maybeEnterFrame(b)
  if (b.isUnused) code push PopUnused
  else code push PopLocal(b.frameSlot)
}
```

Figure 36: Code generation of array constructors, identifiers and binders (`compiler.CodeGen`).

```
private def generateCall(c: Call): Unit = {
  maybeEnterFrame(c)
  if (c.saveReturnPointer) code push SaveReturnPtr(c.returnPointerSlot)
  if (c.saveCurrentContext) code push SaveContext(c.currentContextSlot)
  c.callType match {
    case CallType.Apply => code push Apply(c.isTailCall)
    case CallType.If    => code push If(c.isTailCall)
  }
  if (c.restoreCurrentContext) code push RestoreContext(c.currentContextSlot)
  if (c.restoreReturnPointer) code push RestoreReturnPtr(c.returnPointerSlot)
  maybeLeaveFrame(c)
}
```

Figure 37: Code generation of calls (`compiler.CodeGen`).

4.3.1 The `Compiler` class

The `Compiler` class is shown in Figure 38. It performs the analysis and generates code for all functions in a program, making sure that the assumptions about the order of processing are met.

Recall that there were two assumptions: each pass of a function comes after that pass for all of the nested functions, and comes after the earlier passes for that function. It is easy to see that the code in the `compile` method satisfies both of these assumptions, because the call to `recursiveCompile` comes first and the remaining calls are in the correct order.

When we do the recursive compile, we mustn't forget to check inside array constructors for nested functions. This is why, aside from calling `compile` for function terms, it also calls itself recursively for array constructors. Functions occurring inside arrays will thus be found, as will functions inside arrays inside arrays, and so on.

```
package compiler

import collection.mutable.Stack
import runtime.Bytecode

class Compiler {
  // Code is generated here.
  val codeStack = new Stack[Bytecode]
  def code = codeStack.toArray

  // Compile a function, after recursively compiling all nested functions.
  def compile(function: FunctionTerm): Unit = {
    recursiveCompile(function.body)
    new Nonlocals(function).analyse
    new Liveness(function).analyse
    new SlotAlloc(function).allocate
    new CodeGen(function, codeStack).generate
  }

  // Compile all nested functions in the body.
  def recursiveCompile(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => compile(nested)
      case ArrayCtor(arrayBody)       => recursiveCompile(arrayBody)
      case _                          =>
    }
}
```

Figure 38: The `Compiler` class.

All code generators are passed the `codeStack` value, so they all push their code onto the one stack. Once the program has been compiled the resulting bytecode can be extracted with the `code` method, which converts `codeStack` to the required array.

4.3.2 The `Main` object

The `Main` object in Figure 39 implements an application that reads in a GML program, compiles and executes it, then prints out the contents of the data stack.

Input is read from the file given by the first argument, if there are any command line arguments, otherwise it is read from standard input.

The parsed terms are wrapped in a function term before being compiled. This doesn't change the meaning of the GML program, because evaluating the terms directly is equivalent to applying the function term in an empty environment. Note that if the list of nonlocals for the compiled function is *not* empty, this is a problem. It means some identifiers being referred to are not bound to any value, so in this case we report an error and exit.

We then execute the code for `main` and print out the data stack. The call to `vm.execute` is performed in a `try` block in order to catch type errors and

```
package compiler

import io.Source
import runtime._

object Main {
  def main(args: Array[String]): Unit = {
    // Get the input source.
    val in = if (args.length > 0) Source.fromFile(args(0))
             else Source.fromInputStream(System.in)

    // Parse the input and wrap the resulting terms in a function.
    val main = FunctionTerm(GmlParser(in.getLines.mkString))

    // Compile to bytecode.
    val compiler = new Compiler
    compiler.compile(main)

    // At the top level, it is an error if there are any nonlocals.
    if (main.nonlocals.length > 0) {
      println("Error: unbound identifier(s): "+ main.nonlocals.mkString(" "))
      exit(1)
    } else {
      // Execute the bytecode.
      val vm = new VM(compiler.code)
      try {
        vm.execute(main.address)
      } catch {
        case e =>
          println(e.getMessage)
          // Close off any unfinished arrays.
          while (vm.data.metaStackTop != 0) vm.data.endArray
      }
      println("Data stack:")
      while (vm.data.stackTop > 0) println(vm.data.pop)
    }
  }
}
```

Figure 39: The `Main` object.

underflow errors in the GML programs, which are not detected by the compiler. If an error is caught, we print its message then fix up the data stack before printing it out. We need to do this because the error may have occurred when building an array, in which case the meta-stack would not be empty and the code to print the stack would not work properly.

The next section has some GML examples that you can now try out.

4.3.3 Some GML Examples

Our first example is a standard for toy functional programming languages, the factorial function. Consider the following GML code:

```
{ /self /n
  n 2 lessi
    { 1 }
    { n 1 subi self self apply n muli }
  if
} /fac
```

This code uses *anonymous recursion* to implement a form of the factorial function. The first input, which `self` will be bound to, is a closure that is to be called for the recursive case. The idea is that any time we want to apply `fac`, we should push two copies of it: one that is used by the apply operator, and one that `self` gets bound to. The body of the function compares `n`, and if it is less than two then one is returned. If not then we subtract one from `n` and do the recursive call, remembering to push two copies of the closure, then multiply the result by `n`.

This function can be used as follows:

```
8 fac fac apply
```

The result will be a data stack containing 40320.

This is a little awkward to use—forgetting to mention `fac` twice will lead to a stack underflow or type error. We can add the following code to help with this.

```
{ fac fac apply } /fac
```

The `fac` inside this function refers to the binding we made above. The function applies `fac` to whtaever is on top of the stack, ensuring that the extra copy is passed as required. The binding uses the same name `fac`, which means that this binding shadows the previous one. This doesn't affect the code inside the function, only the code that comes after this definition, so any code after this point can contain:

```
8 fac apply
```

and expect the right result.

The aim of our next example is a generic function that takes an integer `n` and a function `f`. The effect is that `f` is applied `n` times. We again start with a version that uses anonymous recursion, then shadow it with the real version:

```
{ /self /n /f
  n 1 lessi
    {}
    { f apply f n 1 subi self self apply }
  if
} /repeat

{ repeat repeat apply } /repeat
```

If `n` is less than one then we call the identity function which does nothing. Otherwise we apply `f` once then make the recursive call.

For example, we can create an array containing twenty zeroes with:

```
[ { 0 } 20 repeat apply ]
```

Another simple use of `repeat` is to create a Fibonacci sequence on the data stack:

```
{ /n
  { /b /a a b a b addi } n repeat apply
} /fib
```

Here the nested function reads the top two numbers (that is, pops them then pushes them again so as to leave the data stack unchanged) then pushes their sum. Applying the nested function once extends a Fibonacci sequence by one.

For example, the following array contains the first twelve numbers of the Fibonacci sequence starting with two ones:

```
[ 1 1 10 fib apply ]
```

The output from our program is:

```
Data stack:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

5 Rendering

5.1 Points and rays

In GML, points are used to represent positions in 3D space, to represent colors, and can also be used as a general data structure. Figure 40 shows the implementation of a `Point` class, which we shall use for these three purposes. We shall also use it internally in the rendering package to represent the velocity of a ray, which is a vector quantity.[3]

A number of methods implementing vector algebra operations are provided. Most of these are standard, such as dot and cross product. The `dotXZ` is the

[3]Since the scene being rendered is not moving, it doesn't matter that the rays have a speed as well as a direction. We treat the rays as having a velocity rather than just a direction, because it simplifies our calculations later on.

```
package runtime.render

case class Point(x: Double, y: Double, z: Double) {
  import Math.{max, sqrt}

  // vector sum, difference
  def +(p: Point) = Point(x+p.x, y+p.y, z+p.z)
  def -(p: Point) = Point(x-p.x, y-p.y, z-p.z)

  // scalar product, quotient
  def *(s: Double) = Point(x*s, y*s, z*s)
  def /(s: Double) = *(1/s)

  // vector dot product
  def dot(p: Point) = x*p.x + y*p.y + z*p.z

  // vector dot product on XZ plane
  def dotXZ(p: Point) = x*p.x + z*p.z

  // vector dot product with Y-conjugate
  def conjY(p: Point) = x*p.x - y*p.y + z*p.z

  // vector cross product
  def cross(p: Point) = Point(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x)

  // coordinate-wise product
  def times(p: Point) = Point(x*p.x, y*p.y, z*p.z)

  // maximum value of any coordinate
  def maxCoordinate = max(x, max(y, z))

  // length squared
  def squared = dot(this)

  // additive inverse
  def neg = *(-1)

  // normalize
  def unit = this / sqrt(this squared)
}
```

Figure 40: The `Point` class.

```
package runtime.render

case class Ray(origin: Point, velocity: Point) {
  // Where will the ray extend to after the given time?
  def apply(time: Double): Point = origin + velocity * time

  // Return the ray with normalized velocity.
  def unit = Ray(origin, velocity.unit)
}
```

Figure 41: The `Ray` class.

dot product of the vectors projected onto the X-Z plane. It is the same as the dot product, except without the middle term. The "y-conjugate" of a point is a point with the same x and z values, but the y value is negated. The dot product with a y-conjugate is therefore the same as the dot product, except with the middle term negated.

Note that we provide a method for the length squared, but not the length. Calculating the latter requires an expensive `sqrt` operation, so we avoid it as much as possible. In most cases we just want to compare two lengths to see which is longer; for this purpose it suffices to compare the square of the lengths instead.

The `times` method is for dealing with color. If a light of one color shines on a surface of another color, the resulting visible color will be the coordinate-wise product of the two. The `maxCoordinate` method is also for dealing with color; it is used to determine whether a color is so close to black that it can be treated as such.

We will find it convenient to use the `Ray` class, shown in Figure 41. This simple class consists of two point parameters: `origin`, which is a point in 3D space, and a `velocity`, which is a vector. A ray can be thought of as starting at `origin` at time zero and travelling with `velocity` indefinitely. Applying a ray to a time value gives the location of the ray at that time. The `unit` method returns a ray with the same origin and direction, but a speed of one.

5.2 Lights

Each light in GML has its own color, and there are three kinds of light. The different kinds of light are characterised by three behaviours: how the direction towards the light changes as the observer moves, how much the light is attenuated due to the position of the observer, and whether an object at a given distance will block the light reaching an observer.

The `Light` class in Figure 42 captures the behaviour of lights. We use an abstract value for the color, and three abstract methods for the behaviours. The methods correspond directly to the behaviours in the previous paragraph, except that `isBlocked` takes the distance squared as its argument. As mentioned earlier, we use the lengths squared for comparison to save ourselves the expense

```
package runtime.render

sealed abstract class Light {
  // The color of the light.
  val color: Point

  // Return a unit vector pointing towards the light from point.
  def directionFrom(point: Point): Point

  // Attenuation of the light at a point, given the unit ray pointing
  // towards the light.
  def attenuation(ray: Ray): Double

  // Is the light blocked at point by an object at the given distance
  // squared?
  def isBlocked(point: Point, dSqr: Double): Boolean
}
```

Figure 42: The `Light` base class.

```
case class PlainLight(direction: Point, color: Point) extends Light {
  // Unit vector pointing directly towards the light.
  val towards = direction.neg.unit
  def directionFrom(point: Point) = towards
  def attenuation(ray: Ray) = 1
  def isBlocked(point: Point, dSqr: Double) = true
}

case class PointLight(position: Point, color: Point) extends Light {
  def directionFrom(point: Point) = (position - point).unit
  def attenuation(ray: Ray) = 100 / (99 + (position - ray.origin).squared)
  def isBlocked(point: Point, dSqr: Double) = (position - point).squared > dSqr
}
```

Figure 43: The `PlainLight` and `PointLight` classes.

of calculating the square roots.

Plain directional lights and point lights are shown in Figure 43. The definition of `PlainLight` is simple: plain lights are always blocked by an object no matter how far away, they are never attenuated, and the direction to them from a point is a constant, equal to the unit vector in the opposite direction to the direction of the light. We use a `val` member to store the vector towards the light, which means that it will only be calculated once instead of each time we want to know the direction.

The definition of `Pointlight` is almost as simple. Light is blocked by an object if the light source is further away than the object, attenuation is by the formula given in the specification, and the direction from a point is found with vector subtraction.

The behaviour of spotlights is the same as point lights, except that the attenuation is modified by a factor, or may be cut off entirely, depending on

```
import Math.{cos, pow, toRadians}

case class SpotLight(
              override val position: Point,
              at: Point,
              override val color: Point,
              cutoff: Double,
              exponent: Double
           ) extends PointLight(position, color)
{
  // Unit vector pointing directly towards the light.
  val towards = (position - at).unit

  // Cosine of the cutoff angle.
  val cosCutoff = cos(toRadians(cutoff))

  override def attenuation(ray: Ray) = {
    val cos = ray.velocity dot towards
    if (cos < cosCutoff) 0
    else pow(cos, exponent) * super.attenuation(ray)
  }
}
```

Figure 44: The `SpotLight` class.

the angle between the direction from the observer and the direction directly towards the light. We therefore make the `SpotLight` class extend `PointLight` and override the `attenuation` method. The code is in Figure 44.

`SpotLight` has three parameters that its superclass does not. The `at` parameter is a point at which the cone of light is centered. It could be any distance from the light. The `cutoff` parameter is the angle at which the cone of light expands. Outside this cone no light is produced. Inside the cone the rate at which the light decays with increasing angle is controlled by the `exponent` parameter. The parameters in common with `PointLight` require the `override val` prefix because `PointLight` is a case class that already exports these names for *its* parameters.

We store the direction towards the light in a `val` member, as we did with plain lights. We also store the cosine of the cutoff angle, since we can compare the cosines of angles much like we can compare the squares of lengths, although we need to reverse the comparison because cosine is *decreasing* over the range we are interested in. The cosine of the angle is easy to calculate: it is just the dot product of the ray velocity and the `towards` value (we also need to divide by the length of each vector, but here we have arranged things so that these values are always one).

To complete the calculation, `SpotLight` still needs to call the method in the superclass to find out how much the light is attenuated due to the distance from the source. The `super` keyword is used for this purpose.

```
package runtime

sealed abstract class Scene

case class Sphere(surface: ClosureValue) extends Scene
case class Cube(surface: ClosureValue) extends Scene
case class Cylinder(surface: ClosureValue) extends Scene
case class Cone(surface: ClosureValue) extends Scene
case class Plane(surface: ClosureValue) extends Scene

case class Union(left: Scene, right: Scene) extends Scene
case class Intersect(left: Scene, right: Scene) extends Scene
case class Difference(left: Scene, right: Scene) extends Scene

case class RotateX(scene: Scene, degrees: Double) extends Scene
case class RotateY(scene: Scene, degrees: Double) extends Scene
case class RotateZ(scene: Scene, degrees: Double) extends Scene
case class Translate(scene: Scene, dx: Double, dy: Double, dz: Double)
        extends Scene
case class Scale(scene: Scene, rx: Double, ry: Double, rz: Double)
        extends Scene
case class UScale(scene: Scene, r: Double) extends Scene
```

Figure 45: The `Scene` type.

```
import render.{Point, Light}

case class PointValue(point: Point) extends Value {
  override def toString = point.toString
}

case class LightValue(light: Light) extends Value {
  override def toString = light.toString
}

case class SceneValue(scene: Scene) extends Value {
  override def toString = scene.toString
}
```

Figure 46: Values for scene construction (`runtime.Value`).

5.3 Scene construction

We are now in a position to add scene construction to the virtual machine. Figure 45 shows the `Scene` type, which is used to represent constructed scenes at runtime. Each case corresponds directly to one of the GML operators for scene construction, with parameters corresponding to the operator inputs.

Scenes, as well as points and lights are included as values by adding extra cases to the `Value` hierarchy, as shown in Figure 46. We will also add dynamic type checks to the base class to cover these cases. This is shown in Figure 47.

To implement the scene construction operators it is useful to add some more

```
def toPoint: Point = this match {
  case PointValue(p) => p
  case _ => typeError("Point")
}

def toLight: Light = this match {
  case LightValue(l) => l
  case _ => typeError("Light")
}

def toScene: Scene = this match {
  case SceneValue(s) => s
  case _ => typeError("Scene")
}
```

Figure 47: Dynamic type checks for scene construction (`runtime.Value`).

lifting methods, similar to those from section 3.6. These are shown in Figure 48. As before, the letters at the end of the method names stand for the types that the function argument expects.

The operator implementations are shown in Figure 49. Most of the new operators simply construct a point, light or scene, then wrap it in the appropriate `Value`. We could have defined the spotlight operator using a lifting method as well, but since there is only one operator with this signature it is simpler to just define it in a private method.

5.4 Transformations

Scenes are constructed by combining primitive shapes using set operations, and applying *affine* transformations to move the objects and change their shapes. As discussed in the specification, if we use *homogeneous* coordinates then affine transformations become linear and we can represent them using square matrices of real numbers.

The `Transform` class in Figure 50 is used for the representation. The top three rows of the matrix are listed off as parameters, but since the bottom row consists of "0 0 0 1" for every transformation there's no need to include parameters for these. We use an immutable data structure because later on we will need to multiply the same matrix with two or more other matrices—if the matrices were mutable we'd have to think about when the right time to copy would be, and the problem is complex enough already without having to do that.

Matrix multiplication is defined in the usual way, except with the zeros and ones substituted for the bottom row variables. Obviously, we don't need to include the bottom row values when we construct the result.

Transforming a point or vector is modeled by multiplying the transformation vector on its left. For points, the fourth coordinate is treated as one and for vectors it is treated as zero. In practice this means that a point will be moved

```
import render._

def liftC(f: ClosureValue => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toClosure))

def liftP(f: Point => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toPoint))

def liftSS(f: (Scene, Scene) => Value)(data: DataStack): Unit = {
  val r = data.pop.toScene
  val l = data.pop.toScene
  data.push(f(l, r))
}

def liftSR(f: (Scene, Double) => Value)(data: DataStack): Unit = {
  val r = data.pop.toReal
  val s = data.pop.toScene
  data.push(f(s, r))
}

def liftPP(f: (Point, Point) => Value)(data: DataStack): Unit = {
  val p2 = data.pop.toPoint
  val p1 = data.pop.toPoint
  data.push(f(p1, p2))
}

def liftRRR(f: (Double, Double, Double) => Value)(data: DataStack): Unit = {
  val z = data.pop.toReal
  val y = data.pop.toReal
  val x = data.pop.toReal
  data.push(f(x, y, z))
}

def liftSRRR(f: (Scene, Double, Double, Double) => Value)
            (data: DataStack): Unit =
{
  val z = data.pop.toReal
  val y = data.pop.toReal
  val x = data.pop.toReal
  val s = data.pop.toScene
  data.push(f(s, x, y, z))
}
```

Figure 48: Lifting methods for scene construction (`runtime.Operators`).

```
val opTable: Map[String, OpType] = Map(
  ...
  "cone"       -> liftC(s => SceneValue(Cone(s))),
  "cube"       -> liftC(s => SceneValue(Cube(s))),
  "cylinder"   -> liftC(s => SceneValue(Cylinder(s))),
  "difference" -> liftSS((l, r) => SceneValue(Difference(l, r))),
  "getx"       -> liftP(p => RealValue(p.x)),
  "gety"       -> liftP(p => RealValue(p.y)),
  "getz"       -> liftP(p => RealValue(p.z)),
  "intersect"  -> liftSS((l, r) => SceneValue(Intersect(l, r))),
  "light"      -> liftPP((d, c) => LightValue(PlainLight(d, c))),
  "plane"      -> liftC(s => SceneValue(Plane(s))),
  "point"      -> liftRRR((x, y, z) => PointValue(Point(x, y, z))),
  "pointlight" -> liftPP((p, c) => LightValue(PointLight(p, c))),
  "rotatex"    -> liftSR((s, d) => SceneValue(RotateX(s, d))),
  "rotatey"    -> liftSR((s, d) => SceneValue(RotateY(s, d))),
  "rotatez"    -> liftSR((s, d) => SceneValue(RotateZ(s, d))),
  "scale"      -> liftSRRR((s, x, y, z) => SceneValue(Scale(s, x, y, z))),
  "sphere"     -> liftC(s => SceneValue(Sphere(s))),
  "spotlight"  -> doSpotLight,
  "translate"  -> liftSRRR((s, x, y, z) => SceneValue(Translate(s, x, y, z))),
  "union"      -> liftSS((l, r) => SceneValue(Union(l, r))),
  "uscale"     -> liftSR((s, r) => SceneValue(UScale(s, r)))
)

private def doSpotLight(data: DataStack): Unit = {
  val exponent = data.pop.toReal
  val cutoff = data.pop.toReal
  val color = data.pop.toPoint
  val at = data.pop.toPoint
  val position = data.pop.toPoint
  data.push(LightValue(SpotLight(position, at, color, cutoff, exponent)))
}
```

Figure 49: Operators for scene construction (`runtime.Operators`).

by a translation, but a vector will remain unchanged. That is, a translation changes the position of something but doesn't change its direction or speed.

The companion object in Figure 51 contains methods for constructing primitive transformations, including the identity representing the transformation that does nothing. This is the identity matrix in four dimensions (with its bottom row removed), or equivalently can be thought of as the three dimensional identity, which performs no rotation or scaling, combined with a translation of zero. The primitive transformation matrices are taken from the specification.

When we do the ray tracing later on, we shall be firing rays at transformed shapes to determine the boundaries (if any) where the ray enters or leaves a shape. This is done by performing the inverse transformation on the ray, then calculating the intersections with the shape in its original form. We shall refer to the transformed shapes as being in *world* space, and the originals as being in *shape* space. The transformation goes from shape coordinates to world coordinates, and the inverse goes from world coordinates to shape coordinates.

```
package runtime.render

// Represents a 4x4 matrix:
//    [x1 y1 z1 t1]
//    [x2 y2 z2 t2]
//    [x3 y3 z3 t3]
//    [ 0  0  0  1]
case class Transform(
          x1: Double, y1: Double, z1: Double, t1: Double,
          x2: Double, y2: Double, z2: Double, t2: Double,
          x3: Double, y3: Double, z3: Double, t3: Double
        )
{
  def transformRay(ray: Ray) =
    Ray(transformPoint(ray.origin), transformVector(ray.velocity))

  def transformPoint(p: Point) =
    Point(
      x1*p.x + y1*p.y + z1*p.z + t1,
      x2*p.x + y2*p.y + z2*p.z + t2,
      x3*p.x + y3*p.y + z3*p.z + t3
    )

  def transformVector(p: Point) =
    Point(
      x1*p.x + y1*p.y + z1*p.z,
      x2*p.x + y2*p.y + z2*p.z,
      x3*p.x + y3*p.y + z3*p.z
    )

  def *(m: Transform) =
    Transform(
      x1*m.x1 + y1*m.x2 + z1*m.x3,
      x1*m.y1 + y1*m.y2 + z1*m.y3,
      x1*m.z1 + y1*m.z2 + z1*m.z3,
      x1*m.t1 + y1*m.t2 + z1*m.t3 + t1,
      x2*m.x1 + y2*m.x2 + z2*m.x3,
      x2*m.y1 + y2*m.y2 + z2*m.y3,
      x2*m.z1 + y2*m.z2 + z2*m.z3,
      x2*m.t1 + y2*m.t2 + z2*m.t3 + t2,
      x3*m.x1 + y3*m.x2 + z3*m.x3,
      x3*m.y1 + y3*m.y2 + z3*m.y3,
      x3*m.z1 + y3*m.z2 + z3*m.z3,
      x3*m.t1 + y3*m.t2 + z3*m.t3 + t3
    )
}
```

Figure 50: The `Transform` class.

```
object Transform {
  val identity =
    Transform(1, 0, 0, 0,
              0, 1, 0, 0,
              0, 0, 1, 0)

  def rotationX(deg: Double): Transform = {
    val rad = Math.toRadians(deg)
    val sin = Math.sin(rad)
    val cos = Math.cos(rad)
    Transform(1, 0,    0,   0,
              0, cos, -sin, 0,
              0, sin,  cos, 0)
  }

  def rotationY(deg: Double): Transform = {
    val rad = Math.toRadians(deg)
    val sin = Math.sin(rad)
    val cos = Math.cos(rad)
    Transform(cos, 0, sin, 0,
              0,   1, 0,   0,
             -sin, 0, cos, 0)
  }

  def rotationZ(deg: Double): Transform = {
    val rad = Math.toRadians(deg)
    val sin = Math.sin(rad)
    val cos = Math.cos(rad)
    Transform(cos, -sin, 0, 0,
              sin,  cos, 0, 0,
              0,    0,   1, 0)
  }

  def translation(dx: Double, dy: Double, dz: Double) =
    Transform(1, 0, 0, dx,
              0, 1, 0, dy,
              0, 0, 1, dz)

  def dilation(rx: Double, ry: Double, rz: Double) =
    Transform(rx, 0, 0, 0,
              0, ry, 0, 0,
              0, 0, rz, 0)
}
```

Figure 51: The `Transform` companion object.

```
package runtime.render

import Transform._

case class View(s2w: Transform, w2s: Transform) {
  def rayToWorldSpace(ray: Ray) = s2w.transformRay(ray)
  def rayToShapeSpace(ray: Ray) = w2s.transformRay(ray)

  def pointToWorldSpace(p: Point) = s2w.transformPoint(p)
  def pointToShapeSpace(p: Point) = w2s.transformPoint(p)

  def vectorToWorldSpace(p: Point) = s2w.transformVector(p)
  def vectorToShapeSpace(p: Point) = w2s.transformVector(p)

  def rotateX(deg: Double) =
    View(s2w * rotationX(deg), rotationX(-deg) * w2s)

  def rotateY(deg: Double) =
    View(s2w * rotationY(deg), rotationY(-deg) * w2s)

  def rotateZ(deg: Double) =
    View(s2w * rotationZ(deg), rotationZ(-deg) * w2s)

  def translate(dx: Double, dy: Double, dz: Double) =
    View(s2w * translation(dx, dy, dz), translation(-dx, -dy, -dz) * w2s)

  def scale(rx: Double, ry: Double, rz: Double) =
    View(s2w * dilation(rx, ry, rz), dilation(1/rx, 1/ry, 1/rz) * w2s)

  def uScale(r: Double) = scale(r, r, r)
}

object View {
  val identity = View(Transform.identity, Transform.identity)
}
```

Figure 52: The `View` class and companion object.

We will need to change coordinates in both directions, so we will need a representation of both the transformation matrix and its inverse. Since we can easily calculate the inverse of each of the primitive transformations, we can build the transformation and its inverse at the same time. This is more efficient than using an expensive matrix inversion procedure to calculate the inverse after the transformation is fully built.

The `View` class, shown in Figure 52, contains both the transformation matrix, called `s2w`, and its inverse, called `w2s`. The first six methods transform rays, as well as the constituent points and vectors, from shape to world coordinates and from world to shape coordinates. Each of these methods delegates its implementation to the appropriate matrix.

The remaining methods add a primitive transformation to the view and return a new view. Each of these applies the corresponding transform to the

matrix, and the inverse transform to the inverse matrix, using matrix multiplication. For rotations, the inverse is found by negating the angle of rotation. For translations, the inverse is found by negating each component. For scaling, the inverse is found by taking the reciprocal of each component. Isotropic scaling is simply defined in terms of regular scaling.

The identity is defined in a companion object. It is easy to see that this satisfies an important requirement of transforms, namely that the two matrices really are inverses, because the identity transform is its own inverse.

It is useful for us to verify that the requirement is also satisied for all transforms that we build. Another way of stating the requirement is that `s2w * w2s` must be equal to the identity (modulo rounding errors). Assume that this is true for view `v`, and consider the view `v.rotateX(deg)`. The requirement for this view amounts to

```
v.s2w * rotationX(deg) * rotationX(-deg) * v.w2s
```

being the identity. The middle two terms clearly cancel out, leaving

```
v.s2w * v.w2s
```

which is the identity by our above assumption. The same reasoning applies to all other methods of constructing views, therefore they all satisfy the requirement.

This illustrates why the inverse is calculated by multiplying on the left instead of on the right. Matrix multiplication is not commutative, so it is important that we get the order correct. If we had defined `rotateX` with

```
def rotateX(deg: Double) =
  View(s2w * rotationX(deg), w2s * rotationX(-deg))
```

then the formula would be

```
v.s2w * rotationX(deg) * v.w2s * rotationX(-deg)
```

which would not in general be the identity.

5.5 Boundaries and boundary lists

We will need to calculate the boundaries where a ray enters or leaves a shape in the scene. Ultimately, we will only be interested in the first boundary after time zero that is an entry to a shape, but because of the union, intersection and difference operations, it is hard to know in advance which will be the first. We thus keep all of the boundaries in a list, and only choose the first entry after the final list is calculated.

Figure 53 shows the `Boundary` class, which holds information about the intersection of a ray and a shape, and its companion object. The information stored is the time of impact, the point of impact in shape coordinates, the shape itself and the number of the face that was hit. The `Shape` type is defined in section 5.6.2.

A comparison operator is provided for boundaries. This simply compares the impact times, and is used heavily in the code below. Even though we are

```
package runtime.render

case class Boundary(time: Double, shapePoint: Point, shape: Shape, face: Int) {
  def <(that: Boundary) = time < that.time

  def worldPoint = shape.view.pointToWorldSpace(shapePoint)

  def faceCoordinates = shape.faceCoordinates(face, shapePoint)
}

object Boundary {
  type Boundaries = List[Boundary]

  def union(as: Boundaries, bs: Boundaries): Boundaries =
    (as, bs) match {
      case (Nil, _) => bs
      case (_, Nil) => as
      case (a :: as0, b :: bs0) =>
        if (a < b)      a :: difference(as0, bs)
        else if (b < a) b :: difference(bs0, as)
        else            a :: intersect(as0, bs0)
    }

  def intersect(as: Boundaries, bs: Boundaries): Boundaries =
    (as, bs) match {
      case (Nil, _) => Nil
      case (_, Nil) => Nil
      case (a :: as0, b :: bs0) =>
        if (a < b)      difference(bs, as0)
        else if (b < a) difference(as, bs0)
        else            a :: union(as0, bs0)
    }

  def difference(as: Boundaries, bs: Boundaries): Boundaries =
    (as, bs) match {
      case (Nil, _) => Nil
      case (_, Nil) => as
      case (a :: as0, b :: bs0) =>
        if (a < b)      a :: union(as0, bs)
        else if (b < a) intersect(as, bs0)
        else            difference(bs0, as0)
    }
}
```

Figure 53: The `Boundary` class and companion object.

defining comparison, we don't make this extend the `Ordered` trait. For that trait, the comparison operator must be consistent with the definition of `equals`, but this is not the case for our class because boundaries at the same time may not necessarily be on the same shape. So we just define the method without mentioning the trait, and do without the rich interface it provides.

The `worldPoint` method simply returns the `shapePoint` parameter in world coordinates. The `faceCoordinates` method returns the face coordinates of the surface point. Both of these methods delegate their implementation to `Shape`.

The `Boundary` companion object provides methods for working with lists of boundaries. We keep the lists sorted in order of time, and if there are eny boundaries at exactly the same time we arbitrarily pick one of them. This means that the boundaries in the list alternate between entries and exits.

The methods on boundary lists are the union, intersect and difference operations of a "generalised Boolean algebra". Each is defined by a case expression that matches the two list arguments. The base cases, where one of the arguments is `Nil`, can be easily understood if we interpret `Nil` as the empty set. For example, the first case of `union` states that the union of the empty set with any set `bs` is equal to `bs`.

The recursive cases can be understood if we interpret the head of a list as being the set that starts at that entry point and goes on indefinitely, except with the set represented by the tail removed. Effectively, the list "cons" operator means much the same thing as the difference operation, but its first argument is a single entry rather than a list of boundaries.

Consider the `a < b` case in the union operation. The result has `a` as the first entry, because this is the earliest entry into either set. The tail of the list represents all the things that need to be removed after the entry at `a`. For the union operation, this is `as0`, the holes from `as`, minus everything in `bs`. That is, the elements in `bs` are not holes anymore.

Since union is symmetric, the case for `b < a` is the same as above except with the two arguments reversed. In the remaining case we deal with the tails of both lists; the holes in the result are those elements that are holes in both arguments.

A similar line of reasoning holds for the other two methods—the cautious reader may wish to check this. Two other things worth noting are that the resulting lists are in order and have no duplicates, as mentioned above, and that the algorithm always terminates. To see this last point, note that one element is removed from the head of at least one of the lists in each recursive case, so we must run out of elements eventually.

5.6 The `World` class

`World` is the abstract class used by the rendering engine to represent shapes, their combinations and their transformations. This section describes the class and its subclasses, and gives a factory method for construction of worlds.

```
package runtime.render

import runtime._

abstract class World {
  // Return the list of boundaries where this world-space ray
  // crosses shapes in the world.
  def boundaries(ray: Ray): List[Boundary]
}

class Union(val left: World, val right: World) extends World {
  def boundaries(ray: Ray) =
    Boundary.union(left.boundaries(ray), right.boundaries(ray))
}

class Intersect(val left: World, val right: World) extends World {
  def boundaries(ray: Ray) =
    Boundary.intersect(left.boundaries(ray), right.boundaries(ray))
}

class Difference(val left: World, val right: World) extends World {
  def boundaries(ray: Ray) =
    Boundary.difference(left.boundaries(ray), right.boundaries(ray))
}
```

Figure 54: The `World` base case and some subclasses.

5.6.1 The base class and some subclasses

The base class and some of the subclasses are shown in Figure 54. The rendering engine uses the `World` for one purpose: to find the boundaries crossed by a ray. The `boundaries` abstract method takes a world-space ray as an argument and returns the list of boundaries.

For the subclasses shown here, `Union`, `Intersect` and `Difference`, the boundaries are found by getting the boundary lists for the left and right components, then combining them with the appropriate boundary list operation from the previous section.

5.6.2 The `Shape` subclass

The `Shape` class, shown with its companion object in Figure 55, is an abstract base class for shapes. The first thing to notice is that there is a view parameter as well as the closure value for the surface function. This is what distinguishes these values from the `Scene` values we built in section 5.3. These values include the complete transformations of the shapes, whereas in scenes the transformation was determined by the primitive transforms in the enclosing context. The `World` class thus does not have any subclasses to specifically represent the primitive transforms.

Since it is easier to work in shape coordinates when dealing with a shape, the implementation of `boundaries` converts the ray to shape-space then calls

```
package runtime.render

import runtime.ClosureValue

abstract class Shape(val view: View, val surface: ClosureValue) extends World {
  def boundaries(ray: Ray) =
    shapeBoundaries(view.rayToShapeSpace(ray))

  // Same as boundaries, except the argument is in shape space.
  def shapeBoundaries(ray: Ray): List[Boundary]

  // Given a face and a surface point in shape coordinates, return
  // the unit normal in world coordinates.
  def normal(face: Int, surfacePoint: Point): Point = {
    val n = shapeNormal(face, surfacePoint)

    // Choose an independent vector.
    val q = if (n.x != 0 || n.y != 0) Point(0, 0, 1)
            else Point(1, 0, 0)

    // Find two perpendicular tangents in shape coordinates.
    val t1 = n cross q
    val t2 = n cross t1

    // Convert tangents to world space, which preserves their
    // tangent-hood but not their size.
    val w1 = view.vectorToWorldSpace(t1)
    val w2 = view.vectorToWorldSpace(t2)

    // Recover a unit normal in world space.
    (w1 cross w2).unit
  }

  // Given a face and a surface point in shape coordinates, return
  // a normal vector in shape coordinates.
  def shapeNormal(face: Int, surfacePoint: Point): Point

  // Convert a surface point in shape coordinates to face coordinates.
  def faceCoordinates(face: Int, surfacePoint: Point): (Double, Double)

  ...
}

object Shape {
  // Calculate angle in the range 0..1.
  def theta(y: Double, x: Double): Double = {
    val q = Math.atan2(y, x) / (2 * Math.Pi)
    if (q > 0) q else q + 1
  }
}
```

Figure 55: The `Shape` class and companion object.

`shapeBoundaries`, which is implemented in the subclasses.

Aside from finding the boundaries, we will also need to calculate the surface normal vectors of the shape (that is, vectors pointing perpendicularly out from the surface), and find the face coordinates of a surface point. The latter is left as an abstract method for the subclasses to implement. For the former, we can easily calculate the normal in shape space, but we have a problem because affine transformations don't preserve angles so this won't generally be a normal in world space.

The `normal` method uses a couple of pieces of cleverness to get around this problem. First, the cross product of any two independent (that is, non-parallel) vectors is always at right angles to both of the vectors. Second, while affine transformations don't preserve surface normals, they do preserve tangents. So we start by finding tangents to the shape-space normal, defined abstractly, by picking an independent vector and using the cross product operator twice. When transformed to world space these will still be tangents, so when we use the cross product operator once more we get a vector at right angles to the tangents, which is a surface normal in world space. We return the unit of this vector.

The `Shape` companion object provides a method, `theta`, which is from the specification. It is used by several shapes to calculate their face coordinates.

The `Shape` class also defines some protected methods for the subclasses to use when finding the boundaries. These are shown in Figure 56. The `bound` method provides a slightly shorter way of constructing a boundary for the shape. The `univ` method returns a boundary list with just one entry at the origin of the ray and no exits. We make sure this is intersected with our results, so that we discard all boundaries with time less than zero. The `unitDim` method intersects a boundary list with the part of the ray where a chosen coordinate is between zero and one. This is used to cut the ends off cones and cylinders, and to form all of the sides of a cube.

5.6.3 Quadratic roots

We will need to solve quadratic formulas to find the intersections with curved surfaces. This is fairly standard textbook material, so we won't go into it in detail. The code is shown in Figure 57.

The main thing to note is that we distinguish between the case where there is a double root and where there is one root. A double root comes about when a ray just touches a curved surface but doesn't actually enter. A single root comes about if the ray enters a curved surface but never leaves, such as when a ray is parallel to the curved face of a cone.

5.6.4 Primitive shapes

Each of the primitive shapes is implemented as a subclass of `Shape`. These subclasses need to provide the code to return the boundaries of the shape on a ray, the surface normal in shape coordinates at a point, and the face coordinates of a surface point.

```
// Construct a boundary for the given ray in shape space, time and face.
protected def bound(ray: Ray, time: Double, face: Int) =
  Boundary(time, ray(time), this, face)

// Boundary list representing the whole ray.
protected def univ(ray: Ray) =
  List(bound(ray, 0, 0))

// Constrain one dimension by face0 at 0 and face1 at 1.
protected def unitDim(list: List[Boundary], ray: Ray, origin: Double,
                velocity: Double, face0: Int, face1: Int): List[Boundary] =
  if (list == Nil) Nil
  else if (velocity == 0) {
    if (origin < 0 || origin > 1) Nil
    else list
  } else {
    val bound0 = bound(ray, -origin / velocity, face0)
    val bound1 = bound(ray, (1 - origin) / velocity, face1)
    val list0 = if (velocity < 0) {
      List(bound1, bound0)
    } else {
      List(bound0, bound1)
    }
    Boundary.intersect(list0, list)
  }
```

Figure 56: Support methods for shapes (`runtime.render.Shape`).

The `Sphere` class is shown in Figure 58. To find the boundaries of a ray, we need find the times where the distance from the origin is equal to one. Equivalently, the distance squared, given by `ray(time).squared`, must be equal to one. We set the quadratic coefficents appropriately and solve.

If there are two roots, the line intersects the sphere and we need to create two boundaries, one for each root. We remove boundaries with time less than zero by intersecting with `univ(ray)`. If there are no roots, the line does not intersect the sphere and the result is `Nil`. If there is a double root, it means that the line is a tangent to the sphere. We treat this as not entering the sphere at all.

The case of one root can't arise for spheres, because this only occurs if `a` is zero and `b` is nonzero. For spheres, `a` can only be zero if the velocity of the ray is zero, which doesn't make sense. (Even if it did, `b` would also be zero, so we wouldn't get a single root anyway.)

Calculating a surface normal is easy for spheres: it has the same value as the surface point itself. The formulas for face coordinates are taken from the specification, for spheres and also for the other shapes below.

Figure 59 shows the `Cube` class. To get the boundaries, we start with the whole ray and call `unitDim` for each dimension, passing the boundary list from one call to the next. The result from the last call is returned.

There are only six distinct surface normals, one for each face. We store an

```
package runtime.render

object Quadratic {
  sealed abstract class Roots
  case object NoRoots extends Roots
  case class OneRoot(root: Double) extends Roots
  case class DoubleRoot(root: Double) extends Roots
  case class TwoRoots(low: Double, high: Double) extends Roots

  // Return roots of a*t*t + 2*b*t + c.
  def solve(a: Double, b: Double, c: Double): Roots =
    if (a < 0) solvePos(-a, -b, -c)
    else if (a > 0) solvePos(a, b, c)
    else if (b != 0) OneRoot(-c / (2 * b))
    else NoRoots

  // As above but assumes a > 0.  Method to reduce rounding error
  // adapted from "Numerical recipes".
  def solvePos(a: Double, b: Double, c: Double): Roots = {
    val dSqr = b*b - a*c
    if (dSqr < 0) NoRoots
    else if (dSqr == 0) DoubleRoot(-b / a)
    else {
      val d = Math.sqrt(dSqr)
      if (b < 0) {
        val q = d - b
        TwoRoots(c / q, q / a)
      } else if (b > 0) {
        val q = -(d + b)
        TwoRoots(q / a, c / q)
      } else {
        val q = d / a
        TwoRoots(-q, q)
      }
    }
  }
}
```

Figure 57: The `Quadratic` object.

array of these normals, indexed by face, in the `Normal` object inside the method. The method itself just returns the corresponding entry in the array.

Figure 60 shows the cylinder class. Finding the boundaries of the curved surface of cylinders is similar to finding the boundaries of spheres, except that we project everything onto the X-Z plane (that is, we treat the Y-components of the origin and velocity as zero). There is one more subtle difference in the case of no roots: the ray might be *inside* the cylinder, traveling parallel to it. In this case, there will be no roots but the origin will be inside the unit circle in the X-Z plane. We can test for this by checking if `c` is less than zero. If fully inside the cylinder, the result of this part is `univ(ray)`, otherwise it is `Nil`. Before returning the list, the `unitDim` method is called to cut the ends off the cylinder.

Faces 1 and 2, which are the two ends of the cylinder, have constant surface

```
package runtime.render

import runtime.ClosureValue

class Sphere(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) = {
    // Solve ray(time).squared = 1.
    val a = ray.velocity.squared
    val b = ray.origin dot ray.velocity
    val c = ray.origin.squared - 1
    Quadratic.solve(a, b, c) match {
      case Quadratic.TwoRoots(low, high) =>
        val list = List(bound(ray, low, 0), bound(ray, high, 0))
        Boundary.intersect(list, univ(ray))
      case _ => Nil
    }
  }

  def shapeNormal(face: Int, surfacePoint: Point) = surfacePoint

  def faceCoordinates(face: Int, surfacePoint: Point) =
    (Shape.theta(surfacePoint.x, surfacePoint.z), (surfacePoint.y + 1) / 2)
}
```

Figure 58: The `Sphere` class.

normals. For the curved face, calculation of the surface normal is the same as for spheres except projected onto the X-Z plane.

The `Cone` class is shown in Figure 61. Finding the boundaries involves solving similar equations as before, except that the Y-terms are negated in the sums. We do this with the `conjY` method on points. These equations actually describe a double-sided cone, which is shaped like an infinitely large hourglass. The call to `unitDim` at the end of the method slices off most of one side to produce the flat face of the cone, and slices the other side off entirely.

If there are no roots, the cone is not intersected at all.

If there is one root, this means the ray is parallel to one side of the cone, and the root is the location where the ray crosses the other side of the cone. We need to account for two cases: one where we start outside the cone and enter at the root, and the other where we start inside the cone and leave at the root.

If there is a double root, the ray touches the cone at one location. This could happen because the ray brushes the outside of the cone, which we treat as being fully outside the cone, or because the ray is fully inside the cone and travels exactly through the tip where the two sides of the cone meet.

In the case of two roots, there are again two explanations. The ray could be traveling *across* the axis of the cone, in which case it starts outside the cone, enters on one side, then leaves while still on the same side. Alternatively, the ray could be traveling *along* the axis of the cone, in which case it starts inside one side of the cone, leaves the cone in the middle, then enters again on the other side.

```
package runtime.render

import runtime.ClosureValue

class Cube(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) = {
    var list = univ(ray)

    // faces 0, 1: z = 0, 1
    list = unitDim(list, ray, ray.origin.z, ray.velocity.z, 0, 1)

    // faces 2, 3: x = 0, 1
    list = unitDim(list, ray, ray.origin.x, ray.velocity.x, 2, 3)

    // faces 4, 5: y = 1, 0
    unitDim(list, ray, ray.origin.y, ray.velocity.y, 5, 4)
  }

  def shapeNormal(face: Int, surfacePoint: Point) = {
    object Normal {
      val faces: Array[Point] =
        Array(
          Point(0,  0, -1),
          Point(0,  0,  1),
          Point(-1, 0,  0),
          Point(1,  0,  0),
          Point(0,  1,  0),
          Point(0, -1,  0)
        )
    }
    Normal.faces(face)
  }

  def faceCoordinates(face: Int, surfacePoint: Point) = {
    val u = if (face == 2 || face == 3) surfacePoint.z else surfacePoint.x
    val v = if (face == 4 || face == 5) surfacePoint.z else surfacePoint.y
    (u, v)
  }
}
```

Figure 59: The `Cube` class.

```
package runtime.render

import runtime.ClosureValue

class Cylinder(view: View, surface: ClosureValue) extends Shape(view, surface)
{
  def shapeBoundaries(ray: Ray) = {
    // Solve ray(time) dotXZ ray(time) = 1
    val a = ray.velocity dotXZ ray.velocity
    val b = ray.origin dotXZ ray.velocity
    val c = (ray.origin dotXZ ray.origin) - 1
    val list = Quadratic.solve(a, b, c) match {
      case Quadratic.TwoRoots(low, high) =>
        val list0 = List(bound(ray, low, 0), bound(ray, high, 0))
        Boundary.intersect(list0, univ(ray))
      case Quadratic.NoRoots if (c < 0) => univ(ray)
      case _ => Nil
    }
    // face 1, 2: y = 1, 0
    unitDim(list, ray, ray.origin.y, ray.velocity.y, 2, 1)
  }

  def shapeNormal(face: Int, surfacePoint: Point) =
    if (face == 1) Point(0, 1, 0)
    else if (face == 2) Point(0, -1, 0)
    else Point(surfacePoint.x, 0, surfacePoint.z)

  def faceCoordinates(face: Int, surfacePoint: Point) = {
    val u = if (face == 0) Shape.theta(surfacePoint.x, surfacePoint.z)
            else (surfacePoint.x + 1) / 2
    val v = if (face == 0) surfacePoint.y
            else (surfacePoint.z + 1) / 2
    (u, v)
  }
}
```

Figure 60: The `Cylinder` class.

The surface normal is constant for face 1, and for face 0 is found by negating the Y-component of the surface point.

The last primitive shape class is `Plane`, shown in Figure 62. For the boundaries, we first need to check whether the ray is parallel to the plane, in which case it is either fully in or fully out, depending on the Y-component of the origin. Otherwise, we find the boundary by solving a linear equation, and determine whether we are travelling into or out of the plane by looking at the Y-component of the velocity.

The surface normal of a plane is constant at every point, and the face coordinates are just the X and Z coordinates of the surface point.

```
package runtime.render

import runtime.ClosureValue

class Cone(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) = {
    // Solve ray(time) conjY ray(time) = 0
    val a = ray.velocity conjY ray.velocity
    val b = ray.origin conjY ray.velocity
    val c = ray.origin conjY ray.origin
    val list = Quadratic.solve(a, b, c) match {
      case Quadratic.NoRoots => Nil
      case Quadratic.OneRoot(root) =>
        // Parallel to the cone.
        val list0 = List(bound(ray, root, 0))
        if (b < 0) Boundary.intersect(univ(ray), list0) // entering the cone
        else Boundary.difference(univ(ray), list0)      // leaving the cone
      case Quadratic.DoubleRoot(root) =>
        // Tangent to the cone.
        if (a < 0) univ(ray)                            // inside the cone
        else Nil                                        // outside the cone
      case Quadratic.TwoRoots(low, high) =>
        val list0 = List(bound(ray, low, 0), bound(ray, high, 0))
        if (a > 0) Boundary.intersect(univ(ray), list0) // across the cone axis
        else Boundary.difference(univ(ray), list0)      // along the cone axis
    }
    // face 1: y = 1
    unitDim(list, ray, ray.origin.y, ray.velocity.y, 0, 1)
  }

  def shapeNormal(face: Int, surfacePoint: Point) =
    if (face == 1) Point(0, 1, 0)
    else Point(surfacePoint.x, -surfacePoint.y, surfacePoint.z)

  def faceCoordinates(face: Int, surfacePoint: Point) = {
    val u = if (face == 0) Shape.theta(surfacePoint.x, surfacePoint.z)
            else (surfacePoint.x + 1) / 2
    val v = if (face == 0) surfacePoint.y
            else (surfacePoint.z + 1) / 2
    (u, v)
  }
}
```

Figure 61: The `Cone` class.

```
package runtime.render

import runtime.ClosureValue

class Plane(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) =
    if (ray.velocity.y == 0) {
      if (ray.origin.y >= 0) Nil
      else univ(ray)
    } else {
      val list0 = List(bound(ray, -ray.origin.y / ray.velocity.y, 0))
      if (ray.velocity.y < 0) Boundary.intersect(univ(ray), list0)
      else Boundary.difference(univ(ray), list0)
    }

  def shapeNormal(face: Int, surfacePoint: Point) = Point(0, 1, 0)

  def faceCoordinates(face: Int, surfacePoint: Point) =
    (surfacePoint.x, surfacePoint.z)
}
```

Figure 62: The `Plane` class.

5.6.5 The `World` factory.

The `World` companion object, shown in Figure 63, contains a factory method for creating a world out of a scene and an initial view. Before we start rendering, we will call this method with the scene to be rendered and the identity view.

For primitive shapes, we create new instances of the appropriate shape class. For the Boolean operations, we recursively make the worlds for the two parameters then combine them as the appropriate type of world. The same view is passed unchanged to the recursive calls. For the transformations, we recursively make the world for the parameter, but with a view that has been appropriately modified.

The reason we use a separate representation of scenes and worlds is efficiency. We do all of the matrix multiplications in one place, so that we minimize the amount of work to be done. If the matrix multiplications were done during ray tracing, the same computations would effectively be repeated thousands of times which is an enormous and unnecessary cost. On the other hand, if they were done during scene construction we would need to apply *each* transformation to all shapes separately, instead of collapsing all the transformations for a shape into one before applying it. This probably isn't as bad as the first case, but may still be a significant problem.

5.7 Surface properties

The surface of a shape is defined by a surface function, which is a closure value that pops the face number and coordinates off the data stack and pushes the surface properties. To find the surface properties we need to call the virtual

```
object World {
  def make(view: View, scene: Scene): World =
    scene match {
      case Sphere(surface)   => new Sphere(view, surface)
      case Cube(surface)     => new Cube(view, surface)
      case Cylinder(surface) => new Cylinder(view, surface)
      case Cone(surface)     => new Cone(view, surface)
      case Plane(surface)    => new Plane(view, surface)

      case Union(left, right) =>
        new Union(make(view, left), make(view, right))
      case Intersect(left, right) =>
        new Intersect(make(view, left), make(view, right))
      case Difference(left, right) =>
        new Difference(make(view, left), make(view, right))

      case RotateX(scene0, degrees) => make(view.rotateX(degrees), scene0)
      case RotateY(scene0, degrees) => make(view.rotateY(degrees), scene0)
      case RotateZ(scene0, degrees) => make(view.rotateZ(degrees), scene0)
      case Translate(scene0, dx, dy, dz) =>
        make(view.translate(dx, dy, dz), scene0)
      case Scale(scene0, rx, ry, rz) => make(view.scale(rx, ry, rz), scene0)
      case UScale(scene0, r)         => make(view.uScale(r), scene0)
    }
}
```

Figure 63: The `World` companion object.

machine recursively to execute the function—it is recursive because it is called when the virtual machine is running a GML program for which it has reached a `render` operator (more on this later).

The `Properties` class represents surface properties. It is shown in Figure 64 with its companion object. The `surfaceProperties` method in the companion object performs the call to the surface function and returns the result in a `Properties` value.

Since the virtual machine is already running, we need to save the registers before the call and restore them afterwards, otherwise it will get confused when it tries to resume after processing the `render` operator. Doing the call involves pushing the three inputs, then setting the closure context and executing the address, then popping the outputs in reverse order. At the end, the outputs are packaged in a `Properties` value and returned.

5.8 Dealing with surface acne

We have to be careful when dealing with shape boundaries, because of the so-called *surface acne* problem that can result from rounding errors. When a ray hits a shape, we need to fire several rays from the point of intersection to determine how the point is illuminated. If, due to rounding errors, the intersection point ends up slightly inside the shape, the secondary rays will immediately hit the boundary of the shape even though they shouldn't be affected by it.

```
package runtime

import render.Point

case class Properties(color: Point, kd: Double, ks: Double, exp: Double)

object Properties {
  def properties(vm: VM, surface: ClosureValue, face: Int,
                 u: Double, v: Double): Properties =
  {
    // Save registers.
    val savedContext = vm.currentContext
    val savedCodePointer = vm.codePointer
    val savedReturnPointer = vm.returnPointer

    // Perform call.
    vm.data.push(IntValue(face))
    vm.data.push(RealValue(u))
    vm.data.push(RealValue(v))
    vm.currentContext = surface.context
    vm.execute(surface.address)
    val exp = vm.data.pop.toReal
    val ks = vm.data.pop.toReal
    val kd = vm.data.pop.toReal
    val color = vm.data.pop.toPoint

    // Restore registers.
    vm.returnPointer = savedReturnPointer
    vm.codePointer = savedCodePointer
    vm.currentContext = savedContext

    // Return the properties.
    Properties(color, kd, ks, exp)
  }
}
```

Figure 64: The `Properties` class and companion object.

Because of the way we have constructed our boundary lists, any time this rounding problem happens there will be *two* bogus boundaries at the start of the list. One will be the boundary described above, but before that there will be a boundary at time zero, which is left over from intersecting with the `univ(ray)` value (section 5.6.4).

Any time there is a boundary at time zero, it means the origin of the ray was inside a solid shape. Since rays should never get inside solid things to begin with, we know this must be due to a rounding error and we ignore the first pair of boundaries.[4]

The method in Figure 65 performs this step. It is a member of the `Boundary`

[4]Actually, this isn't quite right. It can also happen if the observer is stuck inside a solid object in the scene. The specification doesn't say what to do in such an unfortunate case, but our program at least lets the doomed observer see out of their prison, even if nobody else can see them.

```
  def firstEntry(as: Boundaries): Option[Boundary] =
    as match {
      case a :: _ if (a.time > 0) => Some(a)
      case _ :: _ :: a :: _ => Some(a)
      case _ => None
    }
```

Figure 65: The `firstEntry` method (`runtime.render.Boundary`).

```
package runtime.render

import runtime._

class Render(vm: VM,
             world: World,
             ambient: Point,
             lights: Array[Light],
             maxDepth: Int)
{
  // What is the color if there is no object hit?
  val background = Point(0, 0, 0)

  // Don't consider color components less than this value.
  val cutoff = 0.001

  // Return the square of the distance to the nearest entry, or None
  // if there is no such boundary.
  def fireShadowRay(ray: Ray): Option[Double] =
    Boundary.firstEntry(world.boundaries(ray)) map
            { b => (b.worldPoint - ray.origin).squared }

  ...
}
```

Figure 66: The `Render` class.

companion object. Using this method, we avoid the problem of surface acne without affecting any other part of the image.

5.9 Ray tracing

The main rendering loop and the ray tracing itself will be performed by the `Render` class. The start of this class is shown in Figure 66. The parameters consist of a virtual machine, plus some of the inputs to the render operator. The remaining inputs will be provided to the rendering method, defined later in section 5.11.

It defines two constant values, for the background color and for the minimum color value to consider. When firing rays, if the result is going to be multiplied by a number less than the minimum color value, then we don't bother firing the ray at all and just return black.

The `fireShadowRay` method is used to determine if the light from a particular source reaches a given point without being blocked by another shape. If there are no boundaries on the ray then the light will definitely reach; we return `None` to indicate this result. If there is a boundary then we return its distance squared. This will be passed to the `Light.isBlocked` method from section 5.2.

Before we implement the main ray tracing method, it is helpful to create a class that captures the illumination model. The `Illumination` class in Figure 67 represents the illumination of a boundary point on a ray.

The first part of the class defines several values based on the class parameters. These same values will be used for the specular reflection and the contribution of each light in the scene, so it makes sense to calculate them once for the chosen boundary rather than repeat the calculations multiple times.

The `fromLight` method returns the contribution, specular and diffuse, of a light at the boundary point. The first step is to work out the shadow ray for this light. That is, the unit ray that points towards the light from the boundary. Before doing the rest of the calculation we perform three checks to determine whether there should be any contribution at all.

We first check that the light is above the horizon. That is, the boundary point cannot be in the shadow of the object itself. This is slightly tricky because we don't know whether the normal points inside or outside the surface, but if the incoming velocity vector is on the same side of the surface as the normal, then the shadow velocity vector must be on the opposite side. In other words, the dot product of the incoming velocity and the normal must have the opposite sign to the dot product of the shadow velocity and the normal. Equivalently, the (real) product of these two dot products must be negative, so this is what we check for.

The second check is that the attenuation we calculate is not below the cut-off. Since the end result will be multiplied by the attenuation, if the value is close to zero we might as well return black without performing the rest of the computation.

The last check is done after firing the shadow ray. Here we just check that there is nothing blocking the light.

When calculating the contributions, we again need to account for the fact that the normal vector may be pointing inside the object. The effect would be that the dot products of the normal with the `halfway` vector and with the shadow vector would have the wrong sign. Since we know that both the incident ray and the shadow ray are above the horizon, we know these dot products must be positive, therefore we just take the absolute value.

The main ray tracing method is shown in Figure 68. We first check the depth limit; if the limit is reached we must stop now and return the background color. We then try to find the first entry, and return the background color of none is found.

Having chosen our boundary, we evaluate the surface properties and create an illumination instance. We calculate the light from ambient and specular reflection, but before making the recursive call for specular reflection, we check whether the result would be significant. We don't make the call if the maximum

```
package runtime.render

import Math.{abs, pow}
import runtime.Properties

class Illumination(render: Render,
                   ray: Ray,
                   boundary: Boundary,
                   properties: Properties)
{
  val black = Point(0, 0, 0)
  val point = boundary.worldPoint
  val normal = boundary.shape.normal(boundary.face, boundary.shapePoint)
  val velocityDotNormal = ray.velocity dot normal
  val reflection = Ray(point, ray.velocity - (normal * velocityDotNormal * 2))

  // Illumination at this point from a given light.
  def fromLight(light: Light): Point = {
    val shadowRay = Ray(point, light.directionFrom(point))
    if (velocityDotNormal * (shadowRay.velocity dot normal) >= 0) {
      // The light source is below the horizon.
      return black
    }

    val attenuation = light.attenuation(shadowRay)
    if (attenuation < render.cutoff) {
      // There is not enough light to bother.
      return black
    }

    render.fireShadowRay(shadowRay) match {
      case Some(dSqr) if (light.isBlocked(point, dSqr)) =>
        // The light is blocked by something.
        return black
      case _ =>
    }

    val halfway = (shadowRay.velocity - ray.velocity).unit
    val specular = pow(abs(normal dot halfway), properties.exp) * properties.ks
    val diffuse = abs(normal dot shadowRay.velocity) * properties.kd
    light.color * (specular + diffuse) * attenuation
  }
}
```

Figure 67: The `Illumination` class.

```
// Return the color seen in the direction of this ray,
// assuming we are at the given recursion depth.
def fireRay(ray: Ray, depth: Int): Point =
  if (depth > maxDepth) background
  else Boundary.firstEntry(world.boundaries(ray)) match {
    case None => background
    case Some(boundary) =>
      val (u, v) = boundary.faceCoordinates
      val properties = Properties.properties(vm, boundary.shape.surface,
                                             boundary.face, u, v)
      val illumination = new Illumination(this, ray, boundary, properties)
      val ambientReflection = ambient * properties.kd
      val specularReflection =
        if (properties.ks * properties.color.maxCoordinate < cutoff) {
          background
        } else {
          fireRay(illumination.reflection, depth + 1) * properties.ks
        }
      var lighting = ambientReflection + specularReflection
      for (light <- lights) lighting += illumination.fromLight(light)
      lighting times properties.color
}
```

Figure 68: The `fireRay` method (`runtime.render.Render`).

possible coordinate value of the color is below the cutoff.

We then add up the various components in the illumination model, accumulating them in the `lighting` variable. The final result is the sum of the lighting components times the color of the surface at that point.

5.10 Pixmaps

The output of the program is in the form of a "Portable Pixmap File". Before we get to the rendering loop, we need to write code that writes the pixel colors to a file in this format.

Figure 69 shows the `Pixmap` class, which stores the array of pixels while we are rendering the image then outputs the file when we are finished. The `update` method is called to set a pixel value, or the equivalent assignment syntax is used. The `output` method opens a file with the given name, writes the header and pixel values, then closes the file. The pixel values are converted to the appropriate range before being written.

The pixels are stored in an array of arrays. When the outer array is created we need to create all the inner arrays as well, otherwise when `update` is called a null value will be dereferenced. The two-argument constructor performs this creation for arrays of arrays.

```
package runtime

import render.Point
import java.io.{BufferedOutputStream, FileOutputStream}

class Pixmap(width: Int, height: Int) {
  val header = "P6 "+ width +" "+ height +" 255\n"

  val pixels = new Array[Array[Point]](height, width)

  def update(x: Int, y: Int, color: Point): Unit = {
    pixels(y)(x) = color
  }

  def output(filename: String): Unit = {
    val s = new BufferedOutputStream(new FileOutputStream(filename))
    s.write(header.getBytes)
    for (row <- pixels; pixel <- row) {
      s.write(colorVal(pixel.x))
      s.write(colorVal(pixel.y))
      s.write(colorVal(pixel.z))
    }
    s.close
  }

  // Convert from 0..1 to 0..255.
  def colorVal(v: Double): Int = {
    val scaled = v * 256
    if (scaled < 0) 0
    else if (scaled > 255) 255
    else scaled.toInt
  }
}
```

Figure 69: The `Pixmap` class.

5.11 The `render` operator

The render operator will need to start the main rendering loop, which iterates over all of the pixels in the target image and fires the rays. The rendering loop implementation is in method `render` in class `Render`, shown in Figure 70.

The method first evaluates some constants for use in calculating the ray direction, then creates a `Pixmap` to store the results. The loop itself calls `fireRay`, with an initial depth of 1, on each iteration. We print a "." every eighth row of pixels, so on large examples the user isn't left with a long and mystifying wait before receiving any feedback. We also print a newline after the loop to get all the dots out of the way.

Finally, the method calls the `output` method of the pixmap to write the file.

We can now give the definition of the `render` method in class `VM`, which implements the GML render operator. The code is shown in Figure 71.

The inputs to the operator are first read, in reverse order. The scene value

```
import Math.{tan, toRadians}

// Render the image.
def render(fieldOfView: Double, width: Int, height: Int, filename: String) = {
  val imageWidth = 2 * tan(0.5 * toRadians(fieldOfView))
  val pixelSize = imageWidth / width
  val imageHeight = pixelSize * height
  val imageLeft = -imageWidth / 2
  val imageTop = imageHeight / 2
  val origin = Point(0, 0, -1)
  val pixmap = new Pixmap(width, height)

  // Main loop.
  for (j <- 0 until height) {
    for (i <- 0 until width) {
      val x = imageLeft + (i + 0.5) * pixelSize
      val y = imageTop - (j + 0.5) * pixelSize
      val ray = Ray(origin, Point(x, y, 1).unit)
      pixmap(i, j) = fireRay(ray, 1)
    }
    if (j % 8 == 7) print(".")
  }
  println("")

  // output
  pixmap.output(filename)
}
```

Figure 70: The rendering loop (`runtime.render.Render`).

```
import render.{Render, World, View}

private def render: Unit = {
  val filename = data.pop.toStr
  val height = data.pop.toInt
  val width = data.pop.toInt
  val fieldOfView = data.pop.toReal
  val depth = data.pop.toInt
  val world = World.make(View.identity, data.pop.toScene)
  val lights = data.pop.toArray map { _ toLight }
  val ambient = data.pop.toPoint
  new Render(this, world, ambient, lights, depth).
          render(fieldOfView, width, height, filename)
}
```

Figure 71: The render operator (`runtime.VM`).

must be converted to a `World`, using the `make` factory method. The identity view is passed in to start the process off. Since the `toArray` method just returns an array of values, we need to convert the lights value to an array of lights using a map expression.

After the inputs are read, we create a new instance of `Render` and call the method defined above.

This is the final piece of code for our program. We can now run examples using all of the scene building and rendering functionality of GML.

6 Exercises for the reader

There are probably quite a few ways in which our program can be improved. Any bugs that I am aware of have been fixed, although that's not to say there aren't any others. Kind and/or useful bug reports will be warmly received, of course, and patches even more so.

Some potential improvements are collected here as challenge exercises. Readers who would like to try Scala out for themselves may wish to give one or more of them a go.

Exercise 1
Add an operator `string` that converts any integer to a string containing its decimal representation. Add an operator `append` that appends two strings to form another string. These operators can be used for building a filename inside a function that may be called more than once (for example, to produce an animation consisting of a sequence of still images).

Exercise 2
Extend the lexer to allow a `#include` directive that reads tokens from the included file until the end, then continues reading tokens from after the directive. Recursive includes should also be handled correctly. If a cycle is detected (that is, we reach an include directive for a file that is already being read) the included file should be skipped. Optionally provide a compiler warning when this happens.

Exercise 3
Change the definition of the `Op` class from section 3.3 as follows:

```
case class Op(index: Int) extends Bytecode
```

The `index` parameter is an index into an operator table. Modify the compiler to keep track of the operator index, rather than the operator implementation as it currently does. To execute the bytecode, the virtual machine should look up the implementation in the operator table and run it.

What advantages and/or disadvantages does this design change have for our virtual machine?

Exercise 4
Error handling by the `Main` object is not very sophisticated. For example, out-of-bounds errors for arrays are not printed in a meaningful way. Aside from the type errors that we already report, try to figure out the possible ways a well-formed GML program could cause an exception to be thrown by the virtual machine at runtime. Add code to catch these exceptions and provide the user with some constructive feedback on the error.

Exercise 5
Add add integer parameter to the `Term` base class to hold the line number of the first character in the term. During code generation, create an array of integers parallel to the bytecode array, with each integer being the line number of the term which caused that bytecode to be generated. When a runtime error occurs, the point of execution can be determined from the current value of the code pointer register. Use the information in the array to report the line number of the term that was executing, along with the error message.

Exercise 6
Add a transformation operator `bound` that doesn't change the appearance of the scene at all, but causes a bounding-sphere calculation to be performed. When converting the scene into a world at the start of rendering, a sphere (that is, a center point and radius) that encloses that part of the scene should be found. The sphere *must not* cut off any part of the scene, and ideally it should be as small as possible.

When firing a ray at this scene, a quick check should first be performed to see whether the ray intersects the bounding sphere. If there is no intersection, then there will be no boundaries with the scene inside the sphere and the expensive boundary calculation doesn't need to be performed. If there is an intersection then the boundaries should be calculated as before.

Exercise 7
Add a preprocessing stage to rendering that tries to find good places to automatically add bounding sphere calculations. Better results may be possible if the unions, intersections and differences are rearranged so as to cluster primitive shapes together. This should be done in accordance with the laws of Boolean algebra, to ensure that the image is never affected by the optimisations. Choose a set of benchmarks and try to measure a performance improvement.

Exercise 8
Consider the following slightly contrived GML code:

```
42 /x {{{ ... {{{x}}} ... }}} apply ... apply
```

Each of the nested closures has `x` as a nonlocal; each constructs another closure using `x` then returns, until the innermost one which pushes the value 42. All this constructing of contexts for the nonlocal `x` is somewhat unnecessary, since `x` is bound at the top level and therefore has the constant value 42 wherever it

is used. It is effectively a global constant. The bytecode could equivalently just push the literal value and not use the context at all.

Implement a compiler analysis pass that searches for opportunities to use global constants instead of nonlocal context slots. Hint: in Figure 38 all of the analysis passes are called *after* the call to `recursiveCompile`. In what circumstances would you want a compiler pass to be called before the recursive call?

Exercise 9
If a literal value, or a global constant from the previous exercise, is pushed immediately before an operator such as `sin` or `cos`, the operation can be performed once at compile time and the resulting value substituted into the bytecode. Implement this compiler optimization, and any other extensions to this idea that you can think of.

A The full listing

A.0.1 LICENSE

```
Source code under this directory is Copyright (C) 2010, Mark Brown.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimer in the documentation
      and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

A.1 The `compiler` package

A.1.1 compiler/CodeGen.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import collection.mutable.Stack
import runtime._
```

```
class CodeGen(function: FunctionTerm, code: Stack[Bytecode]) {
  // Generate code for the function.
  def generate: Unit = {
    function.address = code.size
    traverse(function.body)
    function.body.lastOption match {
      case Some(c @ Call(_)) if (c.isTailCall) =>
      case _ => code push Return
    }
  }

  private def traverse(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => generateNested(nested)
      case ArrayCtor(arrayBody)       => generateArrayCtor(arrayBody)
      case id @ Identifier(_)         => generateIdentifier(id)
      case b @ Binder(_)              => generateBinder(b)
      case c @ Call(_)                => generateCall(c)
      case Render                     => code push DoRender
      case Operator(name)             => code push Op(Operators(name))
      case IntLiteral(n)              => code push PushInt(n)
      case RealLiteral(d)             => code push PushReal(d)
      case StringLiteral(s)           => code push PushString(s)
    }

  private def generateNested(nested: FunctionTerm): Unit = {
    maybeEnterFrame(nested)
    code push PushInt(nested.address)
    nested.locations foreach { code push _.bytecode }
    code push MakeClosure(nested.locations.length)
    maybeLeaveFrame(nested)
  }

  private def generateArrayCtor(body: Array[Term]): Unit = {
    code push StartArray
    traverse(body)
    code push EndArray
  }

  private def generateIdentifier(id: Identifier): Unit = {
    code push id.location.bytecode
    maybeLeaveFrame(id)
  }

  private def generateBinder(b: Binder): Unit = {
    maybeEnterFrame(b)
    if (b.isUnused) code push PopUnused
    else code push PopLocal(b.frameSlot)
  }

  private def generateCall(c: Call): Unit = {
    maybeEnterFrame(c)
    if (c.saveReturnPointer) code push SaveReturnPtr(c.returnPointerSlot)
    if (c.saveCurrentContext) code push SaveContext(c.currentContextSlot)
    c.callType match {
      case CallType.Apply => code push Apply(c.isTailCall)
```

```
      case CallType.If    => code push If(c.isTailCall)
    }
    if (c.restoreCurrentContext) code push RestoreContext(c.currentContextSlot)
    if (c.restoreReturnPointer) code push RestoreReturnPtr(c.returnPointerSlot)
    maybeLeaveFrame(c)
  }

  private def maybeEnterFrame(term: Term): Unit =
    if (term.enterFrame) code push EnterFrame(function.frameSize)

  private def maybeLeaveFrame(term: Term): Unit =
    if (term.leaveFrame) code push LeaveFrame(function.frameSize)
}
```

A.1.2 compiler/Compiler.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import collection.mutable.Stack
import runtime.Bytecode

class Compiler {
  // Code is generated here.
  val codeStack = new Stack[Bytecode]
  def code = codeStack.toArray

  // Compile a function, after recursively compiling all nested functions.
  def compile(function: FunctionTerm): Unit = {
    recursiveCompile(function.body)
    new Nonlocals(function).analyse
    new Liveness(function).analyse
    new SlotAlloc(function).allocate
    new CodeGen(function, codeStack).generate
  }

  // Compile all nested functions in the body.
  def recursiveCompile(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => compile(nested)
      case ArrayCtor(arrayBody)       => recursiveCompile(arrayBody)
      case _                          =>
    }
}
```

A.1.3 compiler/GmlLexer.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import util.parsing.combinator.lexical.Lexical
import util.parsing.syntax.StdTokens

class GmlLexer extends Lexical with StdTokens {
  import util.parsing.input.CharArrayReader.EofCh
```

```
  def whitespace: Parser[Any] =
    rep(whitespaceChar | '%' ~ rep(chrExcept('\n', EofCh)))

  def token: Parser[Token] =
    ( name
    | graphic
    | numLiteral
    | stringLiteral
    )

  def name: Parser[Token] =
    letter ~ rep( letter | digit | '-' | '_' ) ^^ {
      case first ~ rest => Identifier(first::rest mkString "")
    }

  def graphic: Parser[Token] =
    ( ('/': Parser[Char]) | '[' | ']' | '{' | '}' ) ^^
           { c => Keyword(c toString) }

  def stringLiteral: Parser[Token] =
    ( '\"' ~> rep(chrExcept('\"', EofCh)) <~ '\"' ^^
           { cs => StringLit(cs mkString "") }
    | '\"' ~> failure("unclosed string literal")
    )

  def numLiteral: Parser[Token] =
    intLiteral ~ optFrac ~ optExp ^^ {
      case i ~ f ~ e => NumericLit(i + f + e)
    }

  def intLiteral: Parser[String] =
    opt('-') ~ rep1(digit) ^^ {
      case None ~ digits => digits mkString ""
      case Some(_) ~ digits => '-'::digits mkString ""
    }

  def optFrac: Parser[String] =
    opt('.' ~> rep1(digit)) ^^ {
      case None => ""
      case Some(frac) => '.'::frac mkString ""
    }

  def optExp: Parser[String] =
    opt((('e': Parser[Char]) | 'E') ~> intLiteral) ^^ {
      case None => ""
      case Some(exp) => "e" + exp
    }
}
```

A.1.4 compiler/GmlParser.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import util.parsing.combinator.syntactical.StdTokenParsers
import util.parsing.syntax.StdTokens
```

```
object GmlParser extends StdTokenParsers {
  type Tokens = StdTokens
  val lexical = new GmlLexer

  def parse(s: String): ParseResult[Array[Term]] =
    phrase(terms)(new lexical.Scanner(s))

  def apply(s: String): Array[Term] =
    parse(s) match {
      case Success(terms, _) => terms
      case e: NoSuccess => throw new Exception("syntax error: "+ e.msg)
    }

  def terms: Parser[Array[Term]] =
    rep(term) ^^ { Array.concat(_) }

  def term: Parser[Term] =
    ( ident                 ^^ { processName(_) }
    | "{" ~> terms <~ "}"   ^^ { FunctionTerm(_) }
    | "[" ~> terms <~ "]"   ^^ { ArrayCtor(_) }
    | "/" ~> ident          ^^ { Binder(_) }
    | numericLit            ^^ { processNum(_) }
    | stringLit             ^^ { StringLiteral(_) }
    )

  import runtime.Operators
  private def processName(name: String): Term =
    if (name == "render") Render
    else if (name == "apply") Call(CallType.Apply)
    else if (name == "if") Call(CallType.If)
    else if (Operators isDefinedAt name) Operator(name)
    else Identifier(name)

  private def processNum(num: String): Term =
    if (num matches "-?[0-9]+") IntLiteral(num.toInt)
    else RealLiteral(num.toDouble)
}
```

A.1.5 compiler/Liveness.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import collection.mutable.HashSet

class Liveness(function: FunctionTerm) {
  // Have we seen a (non-tail) call?
  var seenCall = false

  // Have we seen a call with nonlocal references after it?
  var seenCallWithRestore = false

  // Have we seen a nonlocal reference?
  var seenNonlocal = false
```

```
// Have we seen a nonlocal reference since the last call?
var seenNonlocalSinceCall = false

// Have we seen a local reference?
var seenLocal = false

// Currently live locals.
val liveLocals = new HashSet[String]

// Do the analysis.
def analyse: Unit = {
  function.body.lastOption match {
    case Some(c @ Call(_)) => c.isTailCall = true
    case _ =>
  }
  traverse(function.body)
}

private def traverse(body: Array[Term]): Unit =
  for (term <- body.reverse) term match {
    case nested @ FunctionTerm(_)   => handleNested(nested)
    case ArrayCtor(arrayBody)       => traverse(arrayBody)
    case id @ Identifier(_)         => handleIdentifier(id)
    case b @ Binder(_)              => handleBinder(b)
    case c @ Call(_)                => handleCall(c)
    case _                          =>
  }

private def handleNested(nested: FunctionTerm): Unit = {
  for (name <- nested.parentLocals if !(liveLocals contains name)) {
    // Closure construction is the last to use this local.
    nested.localsToFree = name :: nested.localsToFree
    liveLocals + name
    maybeLeaveFrame(nested)
  }

  if (nested.hasParentNonlocals) {
    seenNonlocal = true
    seenNonlocalSinceCall = true
  }
}

private def handleIdentifier(id: Identifier): Unit =
  if (id.isLocal) {
    if (!(liveLocals contains id.name)) {
      // Last reference to this local.
      id.freeLocal = true
      liveLocals + id.name
      maybeLeaveFrame(id)
    }
  } else {
    seenNonlocal = true
    seenNonlocalSinceCall = true
  }

private def handleBinder(b: Binder): Unit =
  if (liveLocals contains b.name) liveLocals - b.name
```

```
    else b.isUnused = true

  private def handleCall(c: Call): Unit =
    if (!c.isTailCall) {
      // Restore returnPointer if this is the last call.
      if (!seenCall) {
        seenCall = true
        c.restoreReturnPointer = true
        maybeLeaveFrame(c)
      }

      // Save currentContext if there are nonlocal references after the call.
      if (seenNonlocal) c.saveCurrentContext = true

      // Restore currentContext if nonlocals are referenced before the next
      // call.  Free the currentContext slot on the last such occasion.
      if (seenNonlocalSinceCall) {
        seenNonlocalSinceCall = false
        c.restoreCurrentContext = true
        if (!seenCallWithRestore) {
          seenCallWithRestore = true
          c.freeCurrentContext = true
        }
      }
    }

  private def maybeLeaveFrame(term: Term): Unit =
    if (!seenLocal) {
      seenLocal = true
      term.leaveFrame = true
    }
}
```

A.1.6 compiler/Location.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import runtime.{Bytecode, PushLocal, PushNonlocal}

sealed abstract class Location {
  def bytecode: Bytecode
}

case class FrameSlot(slot: Int) extends Location {
  def bytecode = PushLocal(slot)
}

case class ContextSlot(slot: Int) extends Location {
  def bytecode = PushNonlocal(slot)
}
```

A.1.7 compiler/Main.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.
```

```
package compiler

import io.Source
import runtime._

object Main {
  def main(args: Array[String]): Unit = {
    // Get the input source.
    val in = if (args.length > 0) Source.fromFile(args(0))
             else Source.fromInputStream(System.in)

    // Parse the input and wrap the resulting terms in a function.
    val main = FunctionTerm(GmlParser(in.getLines.mkString))

    // Compile to bytecode.
    val compiler = new Compiler
    compiler.compile(main)

    // At the top level, it is an error if there are any nonlocals.
    if (main.nonlocals.length > 0) {
      println("Error: unbound identifier(s): "+ main.nonlocals.mkString(" "))
      exit(1)
    } else {
      // Execute the bytecode.
      val vm = new VM(compiler.code)
      try {
        vm.execute(main.address)
      } catch {
        case e =>
          println(e.getMessage)
          // Close off any unfinished arrays.
          while (vm.data.metaStackTop != 0) vm.data.endArray
      }
      println("Data stack:")
      while (vm.data.stackTop > 0) println(vm.data.pop)
    }
  }
}
```

A.1.8 compiler/Nonlocals.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import collection.mutable.{HashSet, Stack}

class Nonlocals(function: FunctionTerm) {
  // Binders seen.
  val locals = new HashSet[String]

  // Nonlocals allocated.
  val nonlocals = new Stack[String]

  // Do the analysis.
  def analyse: Unit = {
    traverse(function.body)
```

```
    function.nonlocals = nonlocals.toArray
    function.parentLocals ++= nonlocals
  }

  private def traverse(body: Array[Term]): Unit =
    for (term <- body) term match {
      case nested @ FunctionTerm(_)   => handleNested(nested)
      case ArrayCtor(body)            => handleArrayCtor(body)
      case id @ Identifier(name)      => id.isLocal = reference(name)
      case Binder(name)               => locals + name
      case _                          =>
    }

  private def handleNested(nested: FunctionTerm): Unit =
    for (name <- nested.nonlocals if !reference(name)) {
      nested.parentLocals - name
      nested.hasParentNonlocals = true
    }

  private def handleArrayCtor(body: Array[Term]): Unit = {
    val startLocals = locals.clone
    traverse(body)
    locals intersect startLocals
  }

  private def reference(name: String): Boolean = {
    val isLocal = locals contains name
    if (!isLocal && !(function.nonlocalMap contains name)) {
      function.nonlocalMap(name) = nonlocals.size
      nonlocals.push(name)
    }
    isLocal
  }
}
```

A.1.9 compiler/SlotAlloc.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import collection.mutable.HashMap

class SlotAlloc(function: FunctionTerm) {
  // Current slot allocation.
  val frameSlots = new HashMap[String, Int]

  // Number of slots allocated.
  var numFrameSlots = 0

  // Frame slots no longer in use.
  var freeList: List[Int] = Nil

  // Have we entered the frame?
  var frameEntered = false

  // Have we saved returnPointer?
```

```
var savedReturnPointer = false
val nameReturnPointer = "_returnPointer"

// Have we saved currentContext?
var savedCurrentContext = false
val nameCurrentContext = "_currentContext"

// Allocate the slots.
def allocate: Unit = {
  traverse(function.body)
  function.frameSize = numFrameSlots
}

private def traverse(body: Array[Term]): Unit =
  for (term <- body) term match {
    case nested @ FunctionTerm(_)   => handleNested(nested)
    case ArrayCtor(arrayBody)       => traverse(arrayBody)
    case id @ Identifier(_)         => handleIdentifier(id)
    case b @ Binder(_)              => handleBinder(b)
    case c @ Call(_)                => handleCall(c)
    case _                          =>
  }

private def handleNested(nested: FunctionTerm): Unit = {
  // Set the locations of the nested nonlocals.
  nested.locations = nested.nonlocals map { arg =>
    if (nested.parentLocals contains arg) FrameSlot(frameSlots(arg))
    else ContextSlot(function.nonlocalMap(arg))
  }

  // Free all parent locals whose last appearance is in the nested function.
  nested.localsToFree.foreach(freeSlot)
}

private def handleIdentifier(id: Identifier): Unit =
  if (id.isLocal) {
    id.location = FrameSlot(frameSlots(id.name))
    if (id.freeLocal) freeSlot(id.name)
  } else {
    id.location = ContextSlot(function.nonlocalMap(id.name))
  }

private def handleBinder(b: Binder): Unit =
  if (!b.isUnused) {
    maybeEnterFrame(b)
    b.frameSlot = allocateSlot(b.name)
  }

private def handleCall(c: Call): Unit =
  if (!c.isTailCall) {
    maybeEnterFrame(c)

    if (!savedReturnPointer) {
      savedReturnPointer = true
      c.saveReturnPointer = true
      c.returnPointerSlot = allocateSlot(nameReturnPointer)
    }
```

```
      if (c.saveCurrentContext) {
        if (!savedCurrentContext) {
          savedCurrentContext = true
          c.currentContextSlot = allocateSlot(nameCurrentContext)
        } else {
          c.saveCurrentContext = false
        }
      }

      if (c.restoreReturnPointer) {
        c.returnPointerSlot = frameSlots(nameReturnPointer)
        freeSlot(nameReturnPointer)
      }

      if (c.restoreCurrentContext)
        c.currentContextSlot = frameSlots(nameCurrentContext)

      if (c.freeCurrentContext)
        freeSlot(nameCurrentContext)
    }

  private def maybeEnterFrame(term: Term): Unit =
    if (!frameEntered) {
      frameEntered = true
      term.enterFrame = true
    }

  private def allocateSlot(name: String): Int = {
    var slot = -1
    freeList match {
      case firstFree :: rest =>
        slot = firstFree
        freeList = rest
      case Nil =>
        slot = numFrameSlots
        numFrameSlots += 1
    }
    frameSlots(name) = slot
    slot
  }

  private def freeSlot(name: String): Unit = {
    freeList = frameSlots(name) :: freeList
    frameSlots.removeKey(name)
  }
}
```

A.1.10 compiler/Term.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package compiler

import collection.mutable.{HashMap, HashSet}

sealed abstract class Term {
```

```
  // Should we enter the frame before this term?
  var enterFrame = false

  // Should we leave the frame after this term?
  var leaveFrame = false
}

case class FunctionTerm(body: Array[Term]) extends Term {
  // Nonlocal slots.
  var nonlocals: Array[String] = null
  val nonlocalMap = new HashMap[String, Int]

  // Locals from the parent that are referenced here.
  val parentLocals = new HashSet[String]

  // Are nonlocals from the parent referenced here?
  var hasParentNonlocals = false

  // Parent locals to free after the closure is built.
  var localsToFree: List[String] = Nil

  // Where to find nonlocal bindings in the parent.
  var locations: Array[Location] = null

  // Number of frame slots required.
  var frameSize = -1

  // Address of function in generated bytecode.
  var address = -1
}

case class ArrayCtor(body: Array[Term]) extends Term

case class IntLiteral(n: Int) extends Term

case class RealLiteral(d: Double) extends Term

case class StringLiteral(s: String) extends Term

case class Operator(name: String) extends Term

case class Identifier(name: String) extends Term {
  // Is the variable bound locally?
  var isLocal = false

  // Should we free the local after pushing?
  var freeLocal = false

  // Where is the binding stored?
  var location: Location = null
}

case class Binder(name: String) extends Term {
  // True if there are no references to the right.
  var isUnused = false

  // Which frame slot is the value stored in?
```

```
  var frameSlot = -1
}

case object Render extends Term

case class Call(callType: CallType.Value) extends Term {
  // Is this the last term in a function body?
  var isTailCall = false

  // Should we save returnPointer before this call?
  var saveReturnPointer = false

  // Should we restore returnPointer after this call?
  var restoreReturnPointer = false

  // Should we save currentContext before this call?
  var saveCurrentContext = false

  // Should we restore currentContext after this call?
  var restoreCurrentContext = false

  // Should we free the currentContext slot after this call?
  var freeCurrentContext = false

  // In which frame slots are the registers stored?
  var returnPointerSlot = -1
  var currentContextSlot = -1
}

object CallType extends Enumeration { val Apply, If = Value }
```

A.2 The runtime package

A.2.1 runtime/Bytecode.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

sealed abstract class Bytecode

case object DoRender extends Bytecode
case class Apply(isTailCall: Boolean) extends Bytecode
case class If(isTailCall: Boolean) extends Bytecode
case object Return extends Bytecode
case class EnterFrame(size: Int) extends Bytecode
case class LeaveFrame(size: Int) extends Bytecode
case class SaveContext(slot: Int) extends Bytecode
case class RestoreContext(slot: Int) extends Bytecode
case class SaveReturnPtr(slot: Int) extends Bytecode
case class RestoreReturnPtr(slot: Int) extends Bytecode
case class MakeClosure(size: Int) extends Bytecode
case class PushInt(n: Int) extends Bytecode
case class PushReal(d: Double) extends Bytecode
case class PushString(s: String) extends Bytecode
case class PushNonlocal(slot: Int) extends Bytecode
case class PushLocal(slot: Int) extends Bytecode
```

```
case class PopLocal(slot: Int) extends Bytecode
case object PopUnused extends Bytecode
case class Op(op: Operators.OpType) extends Bytecode
case object StartArray extends Bytecode
case object EndArray extends Bytecode
```

A.2.2 runtime/DataStack.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

class DataStack {
  // Data stack.
  var stack = new Array[Value](256)
  var stackTop = 0
  var stackBottom = 0

  def push(value: Value): Unit = {
    stack = ExpandArray.set(stack, stackTop, value)
    stackTop += 1
  }

  def pop: Value = {
    if (stackTop == stackBottom) throw new Exception("gml: stack underflow")
    stackTop -= 1
    stack(stackTop)
  }

  // Meta-stack.
  var metaStack = new Array[Int](8)
  var metaStackTop = 0

  def startArray: Unit = {
    metaStack = ExpandArray.set(metaStack, metaStackTop, stackBottom)
    metaStackTop += 1
    stackBottom = stackTop
  }

  def endArray: Unit = {
    val elements = stack.slice(stackBottom, stackTop)
    stackTop = stackBottom
    if (metaStackTop == 0) throw new Exception("gml: meta stack underflow")
    metaStackTop -= 1
    stackBottom = metaStack(metaStackTop)
    push(ArrayValue(elements))
  }
}
```

A.2.3 runtime/ExpandArray.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

object ExpandArray {
  // Set the value at the index, resizing if needed.
```

```
  def set[T](array: Array[T], index: Int, value: T): Array[T] = {
    val newArray = if (index >= array.length) resize(array, index) else array
    newArray(index) = value
    newArray
  }

  // Double the array size until it includes the index.
  def resize[T](array: Array[T], index: Int): Array[T] = {
    var size = array.length
    if (size == 0) size = 1
    while (size <= index) size *= 2
    val newArray = new Array[T](size)
    array.copyToArray(newArray, 0)
    newArray
  }
}
```

A.2.4 runtime/Frames.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

class Frames {
  var frames = new Array[Value](4096)
  var framePtr = -1

  def enter(size: Int): Unit = {
    framePtr += size
    if (framePtr >= frames.length) frames = ExpandArray.resize(frames, framePtr)
  }

  def leave(size: Int): Unit = {
    framePtr -= size
    if (framePtr < -1) throw new Exception("gml: frame stack underflow")
  }

  def apply(slot: Int): Value =
    frames(framePtr - slot)

  def update(slot: Int, value: Value): Unit =
    frames(framePtr - slot) = value
}
```

A.2.5 runtime/Operators.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

import Math._
import render._

object Operators {
  type OpType = DataStack => Unit

  def apply(name: String): OpType =
```

```
  opTable(name)

def isDefinedAt(name: String): Boolean =
  opTable contains name

val opTable: Map[String, OpType] = Map(
  "acos"       -> liftR(x => RealValue(toDegrees(acos(x)))),
  "addf"       -> liftRR((x, y) => RealValue(x + y)),
  "addi"       -> liftII((x, y) => IntValue(x + y)),
  "asin"       -> liftR(x => RealValue(toDegrees(asin(x)))),
  "clampf"     -> liftR(clamp),
  "cos"        -> liftR(x => RealValue(cos(toRadians(x)))),
  "divf"       -> liftRR((x, y) => RealValue(x / y)),
  "divi"       -> liftII((x, y) => IntValue(x / y)),
  "eqf"        -> liftRR((x, y) => BoolValue(x == y)),
  "eqi"        -> liftII((x, y) => BoolValue(x == y)),
  "false"      -> { _ push(BoolValue(false)) },
  "floor"      -> liftR(x => IntValue(floor(x).toInt)),
  "frac"       -> liftR(frac),
  "get"        -> doGet,
  "length"     -> doLength,
  "lessf"      -> liftRR((x, y) => BoolValue(x < y)),
  "lessi"      -> liftII((x, y) => BoolValue(x < y)),
  "modi"       -> liftII((x, y) => IntValue(x % y)),
  "mulf"       -> liftRR((x, y) => RealValue(x * y)),
  "muli"       -> liftII((x, y) => IntValue(x * y)),
  "negf"       -> liftR(x => RealValue(-x)),
  "negi"       -> liftI(x => IntValue(-x)),
  "real"       -> liftI(x => RealValue(x)),
  "sin"        -> liftR(x => RealValue(sin(toRadians(x)))),
  "sqrt"       -> liftR(x => RealValue(sqrt(x))),
  "subf"       -> liftRR((x, y) => RealValue(x - y)),
  "subi"       -> liftII((x, y) => IntValue(x - y)),
  "true"       -> { _ push(BoolValue(true)) },

  "cone"       -> liftC(s => SceneValue(Cone(s))),
  "cube"       -> liftC(s => SceneValue(Cube(s))),
  "cylinder"   -> liftC(s => SceneValue(Cylinder(s))),
  "difference" -> liftSS((l, r) => SceneValue(Difference(l, r))),
  "getx"       -> liftP(p => RealValue(p.x)),
  "gety"       -> liftP(p => RealValue(p.y)),
  "getz"       -> liftP(p => RealValue(p.z)),
  "intersect"  -> liftSS((l, r) => SceneValue(Intersect(l, r))),
  "light"      -> liftPP((d, c) => LightValue(PlainLight(d, c))),
  "plane"      -> liftC(s => SceneValue(Plane(s))),
  "point"      -> liftRRR((x, y, z) => PointValue(Point(x, y, z))),
  "pointlight" -> liftPP((p, c) => LightValue(PointLight(p, c))),
  "rotatex"    -> liftSR((s, d) => SceneValue(RotateX(s, d))),
  "rotatey"    -> liftSR((s, d) => SceneValue(RotateY(s, d))),
  "rotatez"    -> liftSR((s, d) => SceneValue(RotateZ(s, d))),
  "scale"      -> liftSRRR((s, x, y, z) => SceneValue(Scale(s, x, y, z))),
  "sphere"     -> liftC(s => SceneValue(Sphere(s))),
  "spotlight"  -> doSpotLight,
  "translate"  -> liftSRRR((s, x, y, z) => SceneValue(Translate(s, x, y, z))),
  "union"      -> liftSS((l, r) => SceneValue(Union(l, r))),
  "uscale"     -> liftSR((s, r) => SceneValue(UScale(s, r)))
)
```

```
def liftR(f: Double => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toReal))

def liftI(f: Int => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toInt))

def liftC(f: ClosureValue => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toClosure))

def liftP(f: Point => Value)(data: DataStack): Unit =
  data.push(f(data.pop.toPoint))

def liftRR(f: (Double, Double) => Value)(data: DataStack) = {
  val arg2 = data.pop.toReal
  val arg1 = data.pop.toReal
  data.push(f(arg1, arg2))
}

def liftII(f: (Int, Int) => Value)(data: DataStack): Unit = {
  val arg2 = data.pop.toInt
  val arg1 = data.pop.toInt
  data.push(f(arg1, arg2))
}

def liftSS(f: (Scene, Scene) => Value)(data: DataStack): Unit = {
  val r = data.pop.toScene
  val l = data.pop.toScene
  data.push(f(l, r))
}

def liftSR(f: (Scene, Double) => Value)(data: DataStack): Unit = {
  val r = data.pop.toReal
  val s = data.pop.toScene
  data.push(f(s, r))
}

def liftPP(f: (Point, Point) => Value)(data: DataStack): Unit = {
  val p2 = data.pop.toPoint
  val p1 = data.pop.toPoint
  data.push(f(p1, p2))
}

def liftRRR(f: (Double, Double, Double) => Value)(data: DataStack): Unit = {
  val z = data.pop.toReal
  val y = data.pop.toReal
  val x = data.pop.toReal
  data.push(f(x, y, z))
}

def liftSRRR(f: (Scene, Double, Double, Double) => Value)
            (data: DataStack): Unit =
{
  val z = data.pop.toReal
  val y = data.pop.toReal
  val x = data.pop.toReal
  val s = data.pop.toScene
```

```
    data.push(f(s, x, y, z))
  }

  private def clamp(d: Double): Value =
    RealValue(if (d < 0) 0 else if (d > 1) 1 else d)

  private def frac(d: Double): Value =
    RealValue(d - (if (d < 0) ceil(d) else floor(d)))

  private def doGet(data: DataStack): Unit = {
    val i = data.pop.toInt
    val a = data.pop.toArray
    data.push(a(i))
  }

  private def doLength(data: DataStack): Unit =
    data.push(IntValue(data.pop.toArray.length))

  private def doSpotLight(data: DataStack): Unit = {
    val exponent = data.pop.toReal
    val cutoff = data.pop.toReal
    val color = data.pop.toPoint
    val at = data.pop.toPoint
    val position = data.pop.toPoint
    data.push(LightValue(SpotLight(position, at, color, cutoff, exponent)))
  }
}
```

A.2.6 runtime/Pixmap.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

import render.Point
import java.io.{BufferedOutputStream, FileOutputStream}

class Pixmap(width: Int, height: Int) {
  val header = "P6 "+ width +" "+ height +" 255\n"

  val pixels = new Array[Array[Point]](height, width)

  def update(x: Int, y: Int, color: Point): Unit = {
    pixels(y)(x) = color
  }

  def output(filename: String): Unit = {
    val s = new BufferedOutputStream(new FileOutputStream(filename))
    s.write(header.getBytes)
    for (row <- pixels; pixel <- row) {
      s.write(colorVal(pixel.x))
      s.write(colorVal(pixel.y))
      s.write(colorVal(pixel.z))
    }
    s.close
  }
```

```
  // Convert from 0..1 to 0..255.
  def colorVal(v: Double): Int = {
    val scaled = v * 256
    if (scaled < 0) 0
    else if (scaled > 255) 255
    else scaled.toInt
  }
}
```

A.2.7 runtime/Properties.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

import render.Point

case class Properties(color: Point, kd: Double, ks: Double, exp: Double)

object Properties {
  def properties(vm: VM, surface: ClosureValue, face: Int,
                 u: Double, v: Double): Properties =
  {
    // Save registers.
    val savedContext = vm.currentContext
    val savedCodePointer = vm.codePointer
    val savedReturnPointer = vm.returnPointer

    // Perform call.
    vm.data.push(IntValue(face))
    vm.data.push(RealValue(u))
    vm.data.push(RealValue(v))
    vm.currentContext = surface.context
    vm.execute(surface.address)
    val exp = vm.data.pop.toReal
    val ks = vm.data.pop.toReal
    val kd = vm.data.pop.toReal
    val color = vm.data.pop.toPoint

    // Restore registers.
    vm.returnPointer = savedReturnPointer
    vm.codePointer = savedCodePointer
    vm.currentContext = savedContext

    // Return the properties.
    Properties(color, kd, ks, exp)
  }
}
```

A.2.8 runtime/Scene.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

sealed abstract class Scene
```

```
case class Sphere(surface: ClosureValue) extends Scene
case class Cube(surface: ClosureValue) extends Scene
case class Cylinder(surface: ClosureValue) extends Scene
case class Cone(surface: ClosureValue) extends Scene
case class Plane(surface: ClosureValue) extends Scene

case class Union(left: Scene, right: Scene) extends Scene
case class Intersect(left: Scene, right: Scene) extends Scene
case class Difference(left: Scene, right: Scene) extends Scene

case class RotateX(scene: Scene, degrees: Double) extends Scene
case class RotateY(scene: Scene, degrees: Double) extends Scene
case class RotateZ(scene: Scene, degrees: Double) extends Scene
case class Translate(scene: Scene, dx: Double, dy: Double, dz: Double)
        extends Scene
case class Scale(scene: Scene, rx: Double, ry: Double, rz: Double)
        extends Scene
case class UScale(scene: Scene, r: Double) extends Scene
```

A.2.9 runtime/Value.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

import render.{Point, Light}

sealed abstract class Value {
  def toBoolean: Boolean = this match {
    case BoolValue(b) => b
    case _ => typeError("Boolean")
  }

  def toInt: Int = this match {
    case IntValue(n) => n
    case _ => typeError("Int")
  }

  def toReal: Double = this match {
    case RealValue(d) => d
    case _ => typeError("Real")
  }

  def toStr: String = this match {
    case StringValue(s) => s
    case _ => typeError("String")
  }

  def toArray: Array[Value] = this match {
    case ArrayValue(elements) => elements
    case _ => typeError("Array")
  }

  def toClosure: ClosureValue = this match {
    case c @ ClosureValue(_, _) => c
    case _ => typeError("Function")
  }
```

```
  def toPoint: Point = this match {
    case PointValue(p) => p
    case _ => typeError("Point")
  }

  def toLight: Light = this match {
    case LightValue(l) => l
    case _ => typeError("Light")
  }

  def toScene: Scene = this match {
    case SceneValue(s) => s
    case _ => typeError("Scene")
  }

  private def typeError(s: String): Nothing =
    throw new Exception("type error: expected "+ s +" found "+ this)
}

case class BoolValue(b: Boolean) extends Value {
  override def toString = b.toString
}

case class IntValue(n: Int) extends Value {
  override def toString = n.toString
}

case class RealValue(d: Double) extends Value {
  override def toString = d.toString
}

case class StringValue(s: String) extends Value {
  override def toString = "\""+ s +"\""
}

case class ArrayValue(elements: Array[Value]) extends Value {
  override def toString = "["+ elements.mkString(", ") +"]"
}

case class ClosureValue(address: Int, context: Array[Value]) extends Value {
  override def toString = "{&"+ address +", "+ context.mkString(", ") +"}"
}

case class PointValue(point: Point) extends Value {
  override def toString = point.toString
}

case class LightValue(light: Light) extends Value {
  override def toString = light.toString
}

case class SceneValue(scene: Scene) extends Value {
  override def toString = scene.toString
}
```

A.2.10 runtime/VM.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime

import render.{Render, World, View}

class VM(code: Array[Bytecode]) {
  // Stacks.
  val data = new DataStack
  val frames = new Frames

  // Registers.
  var currentContext: Array[Value] = null
  var codePointer = -1
  var returnPointer = -1

  def execute(address: Int): Unit = {
    codePointer = address
    returnPointer = -1
    while (codePointer >= 0) {
      val bytecode = code(codePointer)
      codePointer += 1
      bytecode match {
        case DoRender               => render
        case Apply(isTailCall)      => callClosure(data.pop, isTailCall)
        case If(isTailCall)         => callIf(isTailCall)
        case Return                 => codePointer = returnPointer
        case EnterFrame(size)       => frames.enter(size)
        case LeaveFrame(size)       => frames.leave(size)
        case SaveContext(slot)      => frames(slot) = ArrayValue(currentContext)
        case RestoreContext(slot)   => currentContext = frames(slot).toArray
        case SaveReturnPtr(slot)    => frames(slot) = IntValue(returnPointer)
        case RestoreReturnPtr(slot) => returnPointer = frames(slot).toInt
        case MakeClosure(size)      => makeClosure(size)
        case PushInt(n)             => data.push(IntValue(n))
        case PushReal(d)            => data.push(RealValue(d))
        case PushString(s)          => data.push(StringValue(s))
        case PushNonlocal(slot)     => data.push(currentContext(slot))
        case PushLocal(slot)        => data.push(frames(slot))
        case PopLocal(slot)         => frames(slot) = data.pop
        case PopUnused              => data.pop
        case Op(op)                 => op(data)
        case StartArray             => data.startArray
        case EndArray               => data.endArray
      }
    }
  }

  private def render: Unit = {
    val filename = data.pop.toStr
    val height = data.pop.toInt
    val width = data.pop.toInt
    val fieldOfView = data.pop.toReal
    val depth = data.pop.toInt
    val world = World.make(View.identity, data.pop.toScene)
```

```
    val lights = data.pop.toArray map { _ toLight }
    val ambient = data.pop.toPoint
    new Render(this, world, ambient, lights, depth).
            render(fieldOfView, width, height, filename)
  }

  private def callClosure(value: Value, isTailCall: Boolean): Unit = {
    val closure = value.toClosure
    if (!isTailCall) returnPointer = codePointer
    codePointer = closure.address
    currentContext = closure.context
  }

  private def callIf(isTailCall: Boolean): Unit = {
    val elseVal = data.pop
    val thenVal = data.pop
    val condition = data.pop.toBoolean
    callClosure(if (condition) thenVal else elseVal, isTailCall)
  }

  private def makeClosure(size: Int): Unit = {
    val context = new Array[Value](size)
    for (i <- (0 until size).reverse) context(i) = data.pop
    data.push(ClosureValue(data.pop.toInt, context))
  }
}
```

A.3 The `runtime.render` package

A.3.1 `runtime/render/Boundary.scala`

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

case class Boundary(time: Double, shapePoint: Point, shape: Shape, face: Int) {
  def <(that: Boundary) = time < that.time

  def worldPoint = shape.view.pointToWorldSpace(shapePoint)

  def faceCoordinates = shape.faceCoordinates(face, shapePoint)
}

object Boundary {
  type Boundaries = List[Boundary]

  def union(as: Boundaries, bs: Boundaries): Boundaries =
    (as, bs) match {
      case (Nil, _) => bs
      case (_, Nil) => as
      case (a :: as0, b :: bs0) =>
        if (a < b)      a :: difference(as0, bs)
        else if (b < a) b :: difference(bs0, as)
        else            a :: intersect(as0, bs0)
    }

  def intersect(as: Boundaries, bs: Boundaries): Boundaries =
```

```
    (as, bs) match {
      case (Nil, _) => Nil
      case (_, Nil) => Nil
      case (a :: as0, b :: bs0) =>
        if (a < b)      difference(bs, as0)
        else if (b < a) difference(as, bs0)
        else            a :: union(as0, bs0)
    }

  def difference(as: Boundaries, bs: Boundaries): Boundaries =
    (as, bs) match {
      case (Nil, _) => Nil
      case (_, Nil) => as
      case (a :: as0, b :: bs0) =>
        if (a < b)      a :: union(as0, bs)
        else if (b < a) intersect(as, bs0)
        else            difference(bs0, as0)
    }

  def firstEntry(as: Boundaries): Option[Boundary] =
    as match {
      case a :: _ if (a.time > 0) => Some(a)
      case _ :: _ :: a :: _ => Some(a)
      case _ => None
    }
}
```

A.3.2 runtime/render/Cone.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime.ClosureValue

class Cone(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) = {
    // Solve ray(time) conjY ray(time) = 0
    val a = ray.velocity conjY ray.velocity
    val b = ray.origin conjY ray.velocity
    val c = ray.origin conjY ray.origin
    val list = Quadratic.solve(a, b, c) match {
      case Quadratic.NoRoots => Nil
      case Quadratic.OneRoot(root) =>
        // Parallel to the cone.
        val list0 = List(bound(ray, root, 0))
        if (b < 0) Boundary.intersect(univ(ray), list0) // entering the cone
        else Boundary.difference(univ(ray), list0)      // leaving the cone
      case Quadratic.DoubleRoot(root) =>
        // Tangent to the cone.
        if (a < 0) univ(ray)                            // inside the cone
        else Nil                                        // outside the cone
      case Quadratic.TwoRoots(low, high) =>
        val list0 = List(bound(ray, low, 0), bound(ray, high, 0))
        if (a > 0) Boundary.intersect(univ(ray), list0) // across the cone axis
        else Boundary.difference(univ(ray), list0)      // along the cone axis
    }
```

```
    // face 1: y = 1
    unitDim(list, ray, ray.origin.y, ray.velocity.y, 0, 1)
  }

  def shapeNormal(face: Int, surfacePoint: Point) =
    if (face == 1) Point(0, 1, 0)
    else Point(surfacePoint.x, -surfacePoint.y, surfacePoint.z)

  def faceCoordinates(face: Int, surfacePoint: Point) = {
    val u = if (face == 0) Shape.theta(surfacePoint.x, surfacePoint.z)
            else (surfacePoint.x + 1) / 2
    val v = if (face == 0) surfacePoint.y
            else (surfacePoint.z + 1) / 2
    (u, v)
  }
}
```

A.3.3 runtime/render/Cube.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime.ClosureValue

class Cube(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) = {
    var list = univ(ray)

    // faces 0, 1: z = 0, 1
    list = unitDim(list, ray, ray.origin.z, ray.velocity.z, 0, 1)

    // faces 2, 3: x = 0, 1
    list = unitDim(list, ray, ray.origin.x, ray.velocity.x, 2, 3)

    // faces 4, 5: y = 1, 0
    unitDim(list, ray, ray.origin.y, ray.velocity.y, 5, 4)
  }

  def shapeNormal(face: Int, surfacePoint: Point) = {
    object Normal {
      val faces: Array[Point] =
        Array(
          Point(0,  0, -1),
          Point(0,  0,  1),
          Point(-1, 0,  0),
          Point(1,  0,  0),
          Point(0,  1,  0),
          Point(0, -1,  0)
        )
    }
    Normal.faces(face)
  }

  def faceCoordinates(face: Int, surfacePoint: Point) = {
    val u = if (face == 2 || face == 3) surfacePoint.z else surfacePoint.x
    val v = if (face == 4 || face == 5) surfacePoint.z else surfacePoint.y
```

```
    (u, v)
  }
}
```

A.3.4 runtime/render/Cylinder.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime.ClosureValue

class Cylinder(view: View, surface: ClosureValue) extends Shape(view, surface)
{
  def shapeBoundaries(ray: Ray) = {
    // Solve ray(time) dotXZ ray(time) = 1
    val a = ray.velocity dotXZ ray.velocity
    val b = ray.origin dotXZ ray.velocity
    val c = (ray.origin dotXZ ray.origin) - 1
    val list = Quadratic.solve(a, b, c) match {
      case Quadratic.TwoRoots(low, high) =>
        val list0 = List(bound(ray, low, 0), bound(ray, high, 0))
        Boundary.intersect(list0, univ(ray))
      case Quadratic.NoRoots if (c < 0) => univ(ray)
      case _ => Nil
    }
    // face 1, 2: y = 1, 0
    unitDim(list, ray, ray.origin.y, ray.velocity.y, 2, 1)
  }

  def shapeNormal(face: Int, surfacePoint: Point) =
    if (face == 1) Point(0, 1, 0)
    else if (face == 2) Point(0, -1, 0)
    else Point(surfacePoint.x, 0, surfacePoint.z)

  def faceCoordinates(face: Int, surfacePoint: Point) = {
    val u = if (face == 0) Shape.theta(surfacePoint.x, surfacePoint.z)
            else (surfacePoint.x + 1) / 2
    val v = if (face == 0) surfacePoint.y
            else (surfacePoint.z + 1) / 2
    (u, v)
  }
}
```

A.3.5 runtime/render/Illumination.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import Math.{abs, pow}
import runtime.Properties

class Illumination(render: Render,
                   ray: Ray,
                   boundary: Boundary,
                   properties: Properties)
```

```
{
  val black = Point(0, 0, 0)
  val point = boundary.worldPoint
  val normal = boundary.shape.normal(boundary.face, boundary.shapePoint)
  val velocityDotNormal = ray.velocity dot normal
  val reflection = Ray(point, ray.velocity - (normal * velocityDotNormal * 2))

  // Illumination at this point from a given light.
  def fromLight(light: Light): Point = {
    val shadowRay = Ray(point, light.directionFrom(point))
    if (velocityDotNormal * (shadowRay.velocity dot normal) >= 0) {
      // The light source is below the horizon.
      return black
    }

    val attenuation = light.attenuation(shadowRay)
    if (attenuation < render.cutoff) {
      // There is not enough light to bother.
      return black
    }

    render.fireShadowRay(shadowRay) match {
      case Some(dSqr) if (light.isBlocked(point, dSqr)) =>
        // The light is blocked by something.
        return black
      case _ =>
    }

    val halfway = (shadowRay.velocity - ray.velocity).unit
    val specular = pow(abs(normal dot halfway), properties.exp) * properties.ks
    val diffuse = abs(normal dot shadowRay.velocity) * properties.kd
    light.color * (specular + diffuse) * attenuation
  }
}
```

A.3.6 runtime/render/Light.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import Math.{cos, pow, toRadians}

sealed abstract class Light {
  // The color of the light.
  val color: Point

  // Return a unit vector pointing towards the light from point.
  def directionFrom(point: Point): Point

  // Attenuation of the light at a point, given the unit ray pointing
  // towards the light.
  def attenuation(ray: Ray): Double

  // Is the light blocked at point by an object at the given distance
  // squared?
  def isBlocked(point: Point, dSqr: Double): Boolean
```

```
}

case class PlainLight(direction: Point, color: Point) extends Light {
  // Unit vector pointing directly towards the light.
  val towards = direction.neg.unit
  def directionFrom(point: Point) = towards
  def attenuation(ray: Ray) = 1
  def isBlocked(point: Point, dSqr: Double) = true
}

case class PointLight(position: Point, color: Point) extends Light {
  def directionFrom(point: Point) = (position - point).unit
  def attenuation(ray: Ray) = 100 / (99 + (position - ray.origin).squared)
  def isBlocked(point: Point, dSqr: Double) = (position - point).squared > dSqr
}

case class SpotLight(
              override val position: Point,
              at: Point,
              override val color: Point,
              cutoff: Double,
              exponent: Double
           ) extends PointLight(position, color)
{
  // Unit vector pointing directly towards the light.
  val towards = (position - at).unit

  // Cosine of the cutoff angle.
  val cosCutoff = cos(toRadians(cutoff))

  override def attenuation(ray: Ray) = {
    val cos = ray.velocity dot towards
    if (cos < cosCutoff) 0
    else pow(cos, exponent) * super.attenuation(ray)
  }
}
```

A.3.7 runtime/render/Plane.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime.ClosureValue

class Plane(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) =
    if (ray.velocity.y == 0) {
      if (ray.origin.y >= 0) Nil
      else univ(ray)
    } else {
      val list0 = List(bound(ray, -ray.origin.y / ray.velocity.y, 0))
      if (ray.velocity.y < 0) Boundary.intersect(univ(ray), list0)
      else Boundary.difference(univ(ray), list0)
    }

  def shapeNormal(face: Int, surfacePoint: Point) = Point(0, 1, 0)
```

```
  def faceCoordinates(face: Int, surfacePoint: Point) =
    (surfacePoint.x, surfacePoint.z)
}
```

A.3.8 runtime/render/Point.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

case class Point(x: Double, y: Double, z: Double) {
  import Math.{max, sqrt}

  // vector sum, difference
  def +(p: Point) = Point(x+p.x, y+p.y, z+p.z)
  def -(p: Point) = Point(x-p.x, y-p.y, z-p.z)

  // scalar product, quotient
  def *(s: Double) = Point(x*s, y*s, z*s)
  def /(s: Double) = *(1/s)

  // vector dot product
  def dot(p: Point) = x*p.x + y*p.y + z*p.z

  // vector dot product on XZ plane
  def dotXZ(p: Point) = x*p.x + z*p.z

  // vector dot product with Y-conjugate
  def conjY(p: Point) = x*p.x - y*p.y + z*p.z

  // vector cross product
  def cross(p: Point) = Point(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x)

  // coordinate-wise product
  def times(p: Point) = Point(x*p.x, y*p.y, z*p.z)

  // maximum value of any coordinate
  def maxCoordinate = max(x, max(y, z))

  // length squared
  def squared = dot(this)

  // additive inverse
  def neg = *(-1)

  // normalize
  def unit = this / sqrt(this squared)
}
```

A.3.9 runtime/render/Quadratic.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

object Quadratic {
```

```
  sealed abstract class Roots
  case object NoRoots extends Roots
  case class OneRoot(root: Double) extends Roots
  case class DoubleRoot(root: Double) extends Roots
  case class TwoRoots(low: Double, high: Double) extends Roots

  // Return roots of a*t*t + 2*b*t + c.
  def solve(a: Double, b: Double, c: Double): Roots =
    if (a < 0) solvePos(-a, -b, -c)
    else if (a > 0) solvePos(a, b, c)
    else if (b != 0) OneRoot(-c / (2 * b))
    else NoRoots

  // As above but assumes a > 0.  Method to reduce rounding error
  // adapted from "Numerical recipes".
  def solvePos(a: Double, b: Double, c: Double): Roots = {
    val dSqr = b*b - a*c
    if (dSqr < 0) NoRoots
    else if (dSqr == 0) DoubleRoot(-b / a)
    else {
      val d = Math.sqrt(dSqr)
      if (b < 0) {
        val q = d - b
        TwoRoots(c / q, q / a)
      } else if (b > 0) {
        val q = -(d + b)
        TwoRoots(q / a, c / q)
      } else {
        val q = d / a
        TwoRoots(-q, q)
      }
    }
  }
}
```

A.3.10 runtime/render/Ray.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

case class Ray(origin: Point, velocity: Point) {
  // Where will the ray extend to after the given time?
  def apply(time: Double): Point = origin + velocity * time

  // Return the ray with normalized velocity.
  def unit = Ray(origin, velocity.unit)
}
```

A.3.11 runtime/render/Render.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import Math.{tan, toRadians}
import runtime._
```

```
class Render(vm: VM,
             world: World,
             ambient: Point,
             lights: Array[Light],
             maxDepth: Int)
{
  // What is the color if there is no object hit?
  val background = Point(0, 0, 0)

  // Don't consider color components less than this value.
  val cutoff = 0.001

  // Return the square of the distance to the nearest entry, or None
  // if there is no such boundary.
  def fireShadowRay(ray: Ray): Option[Double] =
    Boundary.firstEntry(world.boundaries(ray)) map
            { b => (b.worldPoint - ray.origin).squared }

  // Return the color seen in the direction of this ray,
  // assuming we are at the given recursion depth.
  def fireRay(ray: Ray, depth: Int): Point =
    if (depth > maxDepth) background
    else Boundary.firstEntry(world.boundaries(ray)) match {
      case None => background
      case Some(boundary) =>
        val (u, v) = boundary.faceCoordinates
        val properties = Properties.properties(vm, boundary.shape.surface,
                                               boundary.face, u, v)
        val illumination = new Illumination(this, ray, boundary, properties)
        val ambientReflection = ambient * properties.kd
        val specularReflection =
          if (properties.ks * properties.color.maxCoordinate < cutoff) {
            background
          } else {
            fireRay(illumination.reflection, depth + 1) * properties.ks
          }
        var lighting = ambientReflection + specularReflection
        for (light <- lights) lighting += illumination.fromLight(light)
        lighting times properties.color
  }

  // Render the image.
  def render(fieldOfView: Double, width: Int, height: Int, filename: String) = {
    val imageWidth = 2 * tan(0.5 * toRadians(fieldOfView))
    val pixelSize = imageWidth / width
    val imageHeight = pixelSize * height
    val imageLeft = -imageWidth / 2
    val imageTop = imageHeight / 2
    val origin = Point(0, 0, -1)
    val pixmap = new Pixmap(width, height)

    // Main loop.
    for (j <- 0 until height) {
      for (i <- 0 until width) {
        val x = imageLeft + (i + 0.5) * pixelSize
        val y = imageTop - (j + 0.5) * pixelSize
```

```
        val ray = Ray(origin, Point(x, y, 1).unit)
        pixmap(i, j) = fireRay(ray, 1)
      }
      if (j % 8 == 7) print(".")
    }
    println("")

    // output
    pixmap.output(filename)
  }
}
```

A.3.12 runtime/render/Shape.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime.ClosureValue

abstract class Shape(val view: View, val surface: ClosureValue) extends World {
  def boundaries(ray: Ray) =
    shapeBoundaries(view.rayToShapeSpace(ray))

  // Same as boundaries, except the argument is in shape space.
  def shapeBoundaries(ray: Ray): List[Boundary]

  // Given a face and a surface point in shape coordinates, return
  // the unit normal in world coordinates.
  def normal(face: Int, surfacePoint: Point): Point = {
    val n = shapeNormal(face, surfacePoint)

    // Choose an independent vector.
    val q = if (n.x != 0 || n.y != 0) Point(0, 0, 1)
            else Point(1, 0, 0)

    // Find two perpendicular tangents in shape coordinates.
    val t1 = n cross q
    val t2 = n cross t1

    // Convert tangents to world space, which preserves their
    // tangent-hood but not their size.
    val w1 = view.vectorToWorldSpace(t1)
    val w2 = view.vectorToWorldSpace(t2)

    // Recover a unit normal in world space.
    (w1 cross w2).unit
  }

  // Given a face and a surface point in shape coordinates, return
  // a normal vector in shape coordinates.
  def shapeNormal(face: Int, surfacePoint: Point): Point

  // Convert a surface point in shape coordinates to face coordinates.
  def faceCoordinates(face: Int, surfacePoint: Point): (Double, Double)

  // Construct a boundary for the given ray in shape space, time and face.
```

```
  protected def bound(ray: Ray, time: Double, face: Int) =
    Boundary(time, ray(time), this, face)

  // Boundary list representing the whole ray.
  protected def univ(ray: Ray) =
    List(bound(ray, 0, 0))

  // Constrain one dimension by face0 at 0 and face1 at 1.
  protected def unitDim(list: List[Boundary], ray: Ray, origin: Double,
                  velocity: Double, face0: Int, face1: Int): List[Boundary] =
    if (list == Nil) Nil
    else if (velocity == 0) {
      if (origin < 0 || origin > 1) Nil
      else list
    } else {
      val bound0 = bound(ray, -origin / velocity, face0)
      val bound1 = bound(ray, (1 - origin) / velocity, face1)
      val list0 = if (velocity < 0) {
        List(bound1, bound0)
      } else {
        List(bound0, bound1)
      }
      Boundary.intersect(list0, list)
    }
}

object Shape {
  // Calculate angle in the range 0..1.
  def theta(y: Double, x: Double): Double = {
    val q = Math.atan2(y, x) / (2 * Math.Pi)
    if (q > 0) q else q + 1
  }
}
```

A.3.13 runtime/render/Sphere.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime.ClosureValue

class Sphere(view: View, surface: ClosureValue) extends Shape(view, surface) {
  def shapeBoundaries(ray: Ray) = {
    // Solve ray(time).squared = 1.
    val a = ray.velocity.squared
    val b = ray.origin dot ray.velocity
    val c = ray.origin.squared - 1
    Quadratic.solve(a, b, c) match {
      case Quadratic.TwoRoots(low, high) =>
        val list = List(bound(ray, low, 0), bound(ray, high, 0))
        Boundary.intersect(list, univ(ray))
      case _ => Nil
    }
  }

  def shapeNormal(face: Int, surfacePoint: Point) = surfacePoint
```

```
  def faceCoordinates(face: Int, surfacePoint: Point) =
    (Shape.theta(surfacePoint.x, surfacePoint.z), (surfacePoint.y + 1) / 2)
}
```

A.3.14 runtime/render/Transform.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

// Represents a 4x4 matrix:
//    [x1 y1 z1 t1]
//    [x2 y2 z2 t2]
//    [x3 y3 z3 t3]
//    [ 0  0  0  1]
case class Transform(
          x1: Double, y1: Double, z1: Double, t1: Double,
          x2: Double, y2: Double, z2: Double, t2: Double,
          x3: Double, y3: Double, z3: Double, t3: Double
        )
{
  def transformRay(ray: Ray) =
    Ray(transformPoint(ray.origin), transformVector(ray.velocity))

  def transformPoint(p: Point) =
    Point(
      x1*p.x + y1*p.y + z1*p.z + t1,
      x2*p.x + y2*p.y + z2*p.z + t2,
      x3*p.x + y3*p.y + z3*p.z + t3
    )

  def transformVector(p: Point) =
    Point(
      x1*p.x + y1*p.y + z1*p.z,
      x2*p.x + y2*p.y + z2*p.z,
      x3*p.x + y3*p.y + z3*p.z
    )

  def *(m: Transform) =
    Transform(
      x1*m.x1 + y1*m.x2 + z1*m.x3,
      x1*m.y1 + y1*m.y2 + z1*m.y3,
      x1*m.z1 + y1*m.z2 + z1*m.z3,
      x1*m.t1 + y1*m.t2 + z1*m.t3 + t1,
      x2*m.x1 + y2*m.x2 + z2*m.x3,
      x2*m.y1 + y2*m.y2 + z2*m.y3,
      x2*m.z1 + y2*m.z2 + z2*m.z3,
      x2*m.t1 + y2*m.t2 + z2*m.t3 + t2,
      x3*m.x1 + y3*m.x2 + z3*m.x3,
      x3*m.y1 + y3*m.y2 + z3*m.y3,
      x3*m.z1 + y3*m.z2 + z3*m.z3,
      x3*m.t1 + y3*m.t2 + z3*m.t3 + t3
    )
}

object Transform {
```

```
  val identity =
    Transform(1, 0, 0, 0,
              0, 1, 0, 0,
              0, 0, 1, 0)

  def rotationX(deg: Double): Transform = {
    val rad = Math.toRadians(deg)
    val sin = Math.sin(rad)
    val cos = Math.cos(rad)
    Transform(1, 0,    0,   0,
              0, cos, -sin, 0,
              0, sin,  cos, 0)
  }

  def rotationY(deg: Double): Transform = {
    val rad = Math.toRadians(deg)
    val sin = Math.sin(rad)
    val cos = Math.cos(rad)
    Transform(cos, 0, sin, 0,
              0,   1, 0,   0,
             -sin, 0, cos, 0)
  }

  def rotationZ(deg: Double): Transform = {
    val rad = Math.toRadians(deg)
    val sin = Math.sin(rad)
    val cos = Math.cos(rad)
    Transform(cos, -sin, 0, 0,
              sin,  cos, 0, 0,
              0,    0,   1, 0)
  }

  def translation(dx: Double, dy: Double, dz: Double) =
    Transform(1, 0, 0, dx,
              0, 1, 0, dy,
              0, 0, 1, dz)

  def dilation(rx: Double, ry: Double, rz: Double) =
    Transform(rx, 0, 0, 0,
              0, ry, 0, 0,
              0, 0, rz, 0)
}
```

A.3.15 runtime/render/View.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import Transform._

case class View(s2w: Transform, w2s: Transform) {
  def rayToWorldSpace(ray: Ray) = s2w.transformRay(ray)
  def rayToShapeSpace(ray: Ray) = w2s.transformRay(ray)

  def pointToWorldSpace(p: Point) = s2w.transformPoint(p)
  def pointToShapeSpace(p: Point) = w2s.transformPoint(p)
```

```
  def vectorToWorldSpace(p: Point) = s2w.transformVector(p)
  def vectorToShapeSpace(p: Point) = w2s.transformVector(p)

  def rotateX(deg: Double) =
    View(s2w * rotationX(deg), rotationX(-deg) * w2s)

  def rotateY(deg: Double) =
    View(s2w * rotationY(deg), rotationY(-deg) * w2s)

  def rotateZ(deg: Double) =
    View(s2w * rotationZ(deg), rotationZ(-deg) * w2s)

  def translate(dx: Double, dy: Double, dz: Double) =
    View(s2w * translation(dx, dy, dz), translation(-dx, -dy, -dz) * w2s)

  def scale(rx: Double, ry: Double, rz: Double) =
    View(s2w * dilation(rx, ry, rz), dilation(1/rx, 1/ry, 1/rz) * w2s)

  def uScale(r: Double) = scale(r, r, r)
}

object View {
  val identity = View(Transform.identity, Transform.identity)
}
```

A.3.16 runtime/render/World.scala

```
// Copyright (C) 2010 Mark Brown.  See the file LICENSE for details.

package runtime.render

import runtime._

abstract class World {
  // Return the list of boundaries where this world-space ray
  // crosses shapes in the world.
  def boundaries(ray: Ray): List[Boundary]
}

class Union(val left: World, val right: World) extends World {
  def boundaries(ray: Ray) =
    Boundary.union(left.boundaries(ray), right.boundaries(ray))
}

class Intersect(val left: World, val right: World) extends World {
  def boundaries(ray: Ray) =
    Boundary.intersect(left.boundaries(ray), right.boundaries(ray))
}

class Difference(val left: World, val right: World) extends World {
  def boundaries(ray: Ray) =
    Boundary.difference(left.boundaries(ray), right.boundaries(ray))
}

object World {
  def make(view: View, scene: Scene): World =
```

```
    scene match {
      case Sphere(surface)   => new Sphere(view, surface)
      case Cube(surface)     => new Cube(view, surface)
      case Cylinder(surface) => new Cylinder(view, surface)
      case Cone(surface)     => new Cone(view, surface)
      case Plane(surface)    => new Plane(view, surface)

      case Union(left, right) =>
        new Union(make(view, left), make(view, right))
      case Intersect(left, right) =>
        new Intersect(make(view, left), make(view, right))
      case Difference(left, right) =>
        new Difference(make(view, left), make(view, right))

      case RotateX(scene0, degrees) => make(view.rotateX(degrees), scene0)
      case RotateY(scene0, degrees) => make(view.rotateY(degrees), scene0)
      case RotateZ(scene0, degrees) => make(view.rotateZ(degrees), scene0)
      case Translate(scene0, dx, dy, dz) =>
        make(view.translate(dx, dy, dz), scene0)
      case Scale(scene0, rx, ry, rz) => make(view.scale(rx, ry, rz), scene0)
      case UScale(scene0, r)         => make(view.uScale(r), scene0)
    }
}
```