Graceful Patterns for Patterns in Grace

MICHAEL HOMER, Victoria University of Wellington
JAMES NOBLE, Victoria University of Wellington

ACM Reference Format:
Homer, M. and Noble, J. 2012. Graceful Patterns for Patterns in Grace. jn 2, 3, Article 1 (May 2010), 16 pages.

1. INTRODUCTION

Grace is a new object-oriented programming language aimed at education. Here we describe patterns relating to pattern-matching in this language.

2. PATTERNS

Problem	**Pattern**
You want to run different code depending on a literal value	Literal Matching
You want to distinguish the types of objects at runtime	Type Matching
You want to distinguish objects based on their inner values	Destructuring Matching
You want to ignore part of a match	Wildcard Patterns
You want to query whether an object matches a pattern	Match Method
You want to derive new patterns from existing ones	Pattern Combinators
You want to be able to match against specialised criteria	Custom Pattern
You want to match against derived state of objects	Custom Destructuring Pattern
You want to use a derived value as the result of a match	Lambda Pattern

2.1 Pattern Relationships

Many of these patterns are complementary, while others represent different solutions to closely related problems. Match Method is an alternative to the match()case() infrastructure used in most of the other patterns, suitable for simple cases with single patterns, while Literal, Type, and Destructuring Matching directly rely on match()case(). Match Method is however very useful when implementing Custom Pattern and Custom Destructuring Pattern, and is required when using Lambda Pattern.

Literal Matching, Type Matching, and Wildcard Patterns will not be used simultaneously in a particular case, but will often be useful alongside one another. A Literal Match or Self-Matching Object (a special case of Custom Pattern) is often used for exceptional or base cases, while Type Matching or Destructuring Matching is used for other cases and a Wildcard Pattern for the remainder.

Custom Pattern often represents an alternative to Pattern Combinators. While Custom Pattern gives more flexibility in achieving the desired behaviour, it does so at the expense of increased complexity, while using Pattern

Combinators is more readily understood but is limited to the behaviour of the available combinators. While they can also be used together, it will often be clearer to choose one or the other in order to concentrate the specialised behaviours in one place. The same trade-offs apply to Custom Destructuring Pattern, which is a kind of Custom Pattern, but destructuring and combining patterns should usually be separate objects.

3. LITERAL MATCHING

How do you run different pieces of code depending on a value?

Programs often deal with data of built-in literal types like Number and String. Often different behaviour is desired for particular values. You might want to do so for base or special cases in recursion, to deal with numeric flags, or to distinguish string modes.

Forces

—You have a piece of data that could be created from a literal.

—You want to execute different code according to the value of that data.

—Many different values should be distinguished.

Therefore: *Use a match-case statement with literals.*

Grace includes a match-case construct for pattern matching. A number or string literal can be used as a pattern to match against. Many different blocks of code can be provided to execute when different patterns match. The syntax follows Grace's ordinary method request and block syntax.

Example

Suppose we have a rendering system with discrete coordinates that displays data points on the screen. When the x coordinate is zero, one, or the current position of the mouse cursor, we should render the point differently. A match-case statement lets us express this directly using numeric literals:

```
match(x)
  case { 0 −> display.drawCurrent(black) }
  case { 1 −> display.drawCurrent(blue) }
  case { (mouse.x) −> display.drawCurrent(red) }
  case { x' −> display.drawCurrent(getColour x') }
```

Here, when x is zero we draw in black, and similarly for one. Most interesting is when the value is the x-location of the mouse cursor: we parenthesize mouse.x to match against its value, and draw in red. For any other value we obtain the standard colour for the position and display it.

Consequences

—**Benefits.** We can express simply what would otherwise be many nested conditional statements. We have made explicit that we are examining only the value x.

—**Liabilities.** We cannot consider anything but the value of x in deciding what to execute. We cannot match anything that is not a literal.

Discussion

Case statements exist in other languages, and were first proposed by Hoare for Wirth's Algol W [Wirth and Hoare 1966]. The `switch-case` statement in C and Java may operate similarly over numeric data that can be compiled to a jump table when all options are statically known. In most languages, including Grace, the same behaviour can be achieved with multiple nested if statements. For small sets of options – no more than two – an if statement will usually be the correct approach.

Related Patterns

—This technique works because these literals are Self-Matching Objects, a variant of Custom Pattern.

—Values that did not match can use a Wildcard Pattern.

Known Uses

Minigrace, a compiler for Grace, uses Literal Matching during lexing of source code. FlockDB, a distributed graph database written in Scala, uses literal matching with strings heavily to differentiate different input tokens. These systems are described in more detail in Section 12.

4. TYPE MATCHING

How do you run different pieces of code depending on the runtime type of a value?

At times in a program you may have a variable holding a value of one of several types or subtypes. You may need to find out what type it currently holds dynamically and treat the value differently somehow.

Forces

—You have a variable whose value may be of one of several types.

—You want to treat values of different types differently.

—You do not want to include the behaviours in the objects themselves.

Therefore: *Use a match-case statement on types.*

Grace types define patterns that can be used for matching. A match-case statement with types can distinguish between different possible dynamic types of an object at runtime.

Example

Suppose we want to print out some of our data structures to the user for debugging. We want to write one piece of code that returns a string representation for multiple types according to an externally-defined format.

```
match(obj)
  case { s : String −> "String({s})" }
  case { n : Number −> "Number({n})" }
  case { w : GuiWindowObject −> "GUI Window(id: {w.id}, title: {w.title})" }
  case { g : GuiObject −> "General GUI Object" }
  ...
  case { c : Colour −> "Colour({c.asString})" }
```

The first block where the matched object is of the correct type will be evaluated, and no more will be tested.

Consequences

—**Benefits.** We can determine the dynamic type of an object. Our matching blocks follow the syntax of normal blocks and are easily understood. We do not need to modify the objects themselves to obtain the behaviour we require. Objects do not need to care what they may be matched against.

—**Liabilities.** We must know all the types we want to treat differently when we write the match statement. If we add a new type that needs special treatment we must modify our matching code and not solely the new object.

Discussion

Whether Type Matching or dynamic dispatch is appropriate will depend on the task. When the desired behaviour is specific to one particular use compiling all of the implementing code in one place, without complicating the interface of the objects, is likely to be best. If the behaviour is more widely useful, adding it to the objects may be more suitable. When the object code or interface cannot be modified dynamic dispatch is not an option, but Type Matching will still allow extending the behaviour. The Visitor pattern has similar properties to Type Matching when

used over a hierarchy of related objects. It is more difficult to add a new case to a Visitor, so when distinguishing new types is likely to be necessary Type Matching will be a better option.

Related Patterns

—Pattern Combinators align type patterns with derived types.

—Destructuring Matching may use particular type patterns.

—Visitor [Gamma et al. 1995] is a way of specialising behaviour by runtime type.

—Double Dispatch, as found in the Common Lisp Object System and other languages, is another way of specialising behaviour by runtime type.

Known Uses

Minigrace uses Type Matching for pretty-printing of its abstract syntax tree. FlockDB uses Type Matching as part of coercing different values into suitable internal representations.

5. DESTRUCTURING MATCHING

How do you obtain data from an object to use while matching?

Many objects contain other data inside them, or represent several pieces of data as a whole. We may wish either to use this data or to match only objects containing particular values.

Forces

—You have a type that encapsulates data inside.

—You want to treat objects containing particular values differently than others.

—You do not want to provide special methods on the objects for handling different values.

Therefore: *Use a destructuring match.*

A pattern can "destructure" the target of the match, obtaining certain values to examine. These values may be used for matching as well or may be used in the matching block.

Example

Suppose our plotting system has point objects with x and y values inside. When rendering, we need to treat points on the axes differently lest they be obscured by the axis itself.

```
type Point = {
  x −> Number
  y −> Number
  extract −> [Number, Number]
}
...
match(pt)
  case { p : Point(0, y) −> drawPixel(1, y) }
  case { p : Point(x, 0) −> drawPixel(x, 1) }
  case { p : Point(x, y) −> drawPixel(x, y) }
  case { nonPoint −> displayError "{nonPoint} is not a point" }
```

Any type containing an extract method supports destructuring matches against it. When we use a destructuring pattern like Point(0, y), match-case first tries to match the target of the match against the Point pattern. If it matches, we then try to match the extracted values against any subpatterns: here, the first value (the x coordinate) against zero. If that matches, the block will be executed with any remaining parameters bound to the corresponding extracted values. Here, points on the y axis are drawn slightly to the right, but at the same height; points on neither axis are drawn where they are; and we report an error when the target of the match is not a Point at all.

Consequences

—**Benefits.** We can distinguish a certain subset of objects of a type for processing, with clear conditions. We do not have nested conditionals or application-specific modifications to our objects to obtain this behaviour. The extracted values are automatically available for use with a concise syntax.

—**Liabilities.** We must repeat the "Point" name for each case. The target of the match must support destructuring. Behaviour depending on internal state may be more suitable inside the objects themselves.

Discussion

Destructured data elements will usually be available through methods on the object itself as well. If there is no matching against values or only a few of the values are used then non-destructuring Type Matching may be simpler. For particular very general cases implementing methods with the right behaviour on the objects themselves may be suitable. When there are few distinguished cases or much of the behaviour is the same for different cases then simple if-then statements inside a more general match may be clearer, although destructuring will be more consistent with other match statements.

Related Patterns

—Wildcard Patterns allow ignoring destructured parts.

—A Custom Destructuring Pattern implements destructuring with user-defined behaviour.

Known Uses

Minigrace uses Destructuring Matching for pretty-printing of its abstract syntax tree. Breeze, a library for numeric processing and AI written in Scala, uses Destructuring Matching extensively for extracting the values encapsulated in its objects.

6. WILDCARD PATTERNS

How do you ignore objects while matching?

A particular value may be unimportant during a match and should be disregarded. When matching against a type we may only care to know the type, and not want to use the value afterwards. How can a value be ignored?

Forces

—You are matching against a pattern, but don't want to use one or more of the values the pattern provides.

Therefore: *Use a wildcard pattern.*

A wildcard pattern is irrefutable and always matches, but does not bind a name. The matching value is discarded. Wildcard patterns are written _, the same name used for unused parameters in blocks and methods.

Example

Suppose that when matching values we want to include a "fall-through" case, to run when none of the other patterns match. We can use a wildcard pattern for this.

```
match(n)
  case { 0 −> 1 }
  case { 1 −> 1) }
  case { _ −> fib(x−1) + fib(x − 2) }
```

Our fall-through case could raise an error, or perform a default behaviour.

Suppose that when matching a Point we only care about the x value, and want to disregard the y and the point itself. We also want to report an error when given a Point3D, but don't care about the value itself when doing so.

```
match(pt)
  case { _ : Point3D −> displayError "3D points are unsupported" }
  case { _ : Point(x, _) −> setPosition(x) }
  case { _ : displayError "Expected a point" }
```

The wildcard values always match, but have no further effect.

Consequences

—**Benefits.** We can clearly indicate which values we are concerned with and explicitly disregard the others. We do not need to choose names that we will not use but that might conflict with other names in scope or that we may define later.

—**Liabilities.** We may want to use the value during debugging, but it will not be available. If we later want to use the value we must update the pattern.

Discussion

Using a wildcard in a pattern is equivalent to using a variable that is then never referenced. In some cases specifying that variable name will help to make the code clearer, even though it is never used. The name can document why the value is unused, which may help the reader.

Related Patterns

—Type Matching is another kind of irrefutable match for a particular type.

—Destructuring Matching will often involve Wildcard Patterns.

7. MATCH METHOD

How do you determine whether something matches without entering a new block?

Sometimes we want to see whether something matches without using match-case. How can we find this out?

Forces

—You want to match against a particular single pattern.

Therefore: *Use the match method.*

Pattern-matching in Grace is built around the match method. Any object may have a match method. Such an object is known as a *Pattern*. The match method on a Pattern returns a MatchResult object, which is an extension of the Boolean type.

Example

Suppose we want to know whether a value obtained from user input is a number or not, without entering a match-case block. We could use a degenerate match-case:

```
def isNumber = match(val) case {_ : Number −> true } case { _ : false }
```

Better is to use the match method directly.

```
if (Number.match(val)) then {
  ...
}
```

The return value of match may be treated as a Boolean. It also contains other information for advanced matching.

Suppose we allow querying over lists of heterogeneous data, returning those values meeting some criterion. We can use a pattern and the match method to test our data and accumulate the successful matches.

```
method filter(data : List)by(pat : Pattern) {
  def values = list.new // To accumulate selected values.
  for (data) do { v ->
    def m = pat.match(v)
    if (m) then {
      values.add(m.result)
    }
  }
  values
}
```

Consequences

—**Benefits.** We can find out whether a pattern matches without using the full match-case structure.

—**Liabilities.** If we later add more distinguished patterns we will need to refactor our code to use match-case.

Discussion

Match Method is the primary access to the pattern matching system itself, and will often be used in methods or objects that accept a pattern as an argument. Using match()case() with such a pattern is more complicated and harder to follow than requesting match(), even when multiple patterns are in use. For local patterns or type patterns the choice between Match Method and Type Matching may be determined by aesthetics and consistency.

Related Patterns

—Custom Pattern includes a user-defined Match Method to implement specialised behaviour.

—Custom Destructuring Pattern includes a user-defined Match Method with the MatchResult's bindings filled in.

—Lambda Pattern sets the SuccessfulMatch.result value.

Known Uses

FlockDB uses Match Method for almost all its access to pattern-matching, and implements equality methods in this way. Minigrace uses Match Method for type tests during pretty-printing that have only two distinguished cases.

8. PATTERN COMBINATORS

How do you combine multiple patterns into one?

Sometimes we care that an object meets not just one pattern, but multiple patterns simultaneously. Other times we want it to match any of a number of patterns. How can we do that?

Forces

—You want to use more than one pattern as part of a single match.

Therefore: *Use pattern combinators.*

A pattern combinator modifies the effect of one or more patterns. Some combinators include the AndPattern combinator & and the OrPattern combinator |.

Example

Suppose that we have patterns OddNumber and Negative. We want to treat numbers that are both odd and negative in a specific way, and special-case zero and one.

```
match(val)
  case { n : OddNumber & Negative −> handleOddNegative n }
  case { 0 | 1 −> handleSmall }
  case { n : Number −> handleNormal n }
```

0 | 1 matches if either the pattern 0 or the pattern 1 matches. OddNumber & Negative matches only if **both** OddNumber and Negative match.

Consequences

—**Benefits.** We can use existing patterns to make new patterns in a standard way. Anybody who understands the existing patterns will understand our constructed patterns.

—**Liabilities.** Not all combinators can pass destructured values through. Some combinators may be obscure.

Discussion

The & and | combinators align with how these operators are used for derived types. The untagged variant type String | Number includes the same values as are matched by the pattern String | Number.

When combinators are used with destructuring match special care is required. While the & combinator collects and retains the bindings from destructuring, the | combinator does not. With the | combinator we do not know which sub-pattern matched, and consequently which names have been bound. The example below shows this:

```
match(o)
  case { Point(0, y) & ColouredThing(col) −>
    print "{col} point on the y axis at {y}" }
  case { Point(0, y) | ColouredThing(col) −>
    // Neither y nor col are defined here }
```

Other combinators may have similar effects. In some cases, a Custom Destructuring Pattern will be able to give the required effect when a different behaviour is desired, but pattern-matching may not be suitable at all when default values or similar functionalities are needed.

Related Patterns

—Type Matching is a common place to use combinators.

—Custom Pattern is another way of creating derived patterns.

Known Uses

Minigrace uses Pattern Combinators for many type-based matches where multiple cases have the same associated behaviour, and the Or Pattern avoids repeating this code.

9. CUSTOM PATTERN

How do you define a pattern with customised behaviour?

Built-in or automatic patterns may not suffice for our code. We may need to match objects in a way that those patterns do not support. How can we use match-case in this situation?

Forces

—You want to match against an object in a way that the built-in patterns from types and literals do not allow.

Therefore: *Implement a custom pattern.*

A Pattern is an object with a method match(o : Object) −> MatchResult. We can define an object with this method with any semantics we require.

Example

Suppose that when displaying data in a table we want to display odd and even rows with a different background colour. We can create an OddNumber pattern:

```
def OddNumber = object {
  method match(o : Object) −> MatchResult {
    match(o)
      case { n : Number −>
        if ((n % 2) == 1) then {
          return SuccessfulMatch.new(o, [])
        }
      }
    FailedMatch.new(o)
  }
}
```

We can then use our pattern in a match:

```
for (rows) do {row−>
  match(row.number)
    case { n : OddNumber −> drawRow(row, grey) }
    case { n : Number −> drawRow(row, white) }
}
```

Self-Matching Object

A data object may also be a custom pattern itself, in which case it makes sense for it to match itself. The Self-Matching Object variant may be implemented with match method using == comparison.

Consequences

—**Benefits.** We can use a pattern in a match with exactly the semantics we require.

—**Liabilities.** A custom pattern may need to be updated when the objects it matches are changed.

Discussion

While a custom pattern can perform any test that can be written in code, it will not always be the best solution to a problem. Simple if-then tests, dynamic dispatch, or ordinary match-case may be more appropriate. Custom Pattern will be suitable when defining new libraries of code where a subset of objects should be distinguished, or where code already uses match-case heavily and fitting the new pattern in with existing code makes it clearer. Self-Matching Object is suitable for immutable data objects, but is usually not the best option otherwise.

Related Patterns

—Match Method is how the custom matching behaviour is used.

Known Uses

Breeze uses Custom Pattern extensively, with patterns for almost all of its public data types, many of which are Self-Matching Objects. Minigrace implements the built-in patterns of Grace using Custom Pattern.

10. CUSTOM DESTRUCTURING PATTERN

How do you define a destructuring pattern with customised behaviour?

When performing a destructuring match we may want to use derived values, or match against data that the standard pattern does not expose. How can we create a pattern to do that?

Forces

—You want to perform a destructuring match, using extracted data a standard pattern does not provide.

—You want to perform a destructuring match using derived data.

Therefore: *Define a custom pattern with bindings.*

A SuccessfulMatch object may contain a set of bindings, used for destructuring matching. The default pattern for a type uses the extract method on that type to obtain bindings, if defined, but we can set our own bindings in our match method.

Example

Suppose that when plotting points we want to mark those whose distance from the origin matches particular values. We can write a custom destructuring pattern to provide this:

```
def DistanceFromOrigin = object {
  method match(o : Object) −> MatchResult {
    match(o)
      case { _ : Point(x,y) −>
        return SuccessfulMatch.new(o, [((x^2) + (y^2)).sqrt])
      }
    FailedMatch.new(o)
  }
}
```

We can then use our pattern in a match:

```
match(pt)
  case { p : DistanceFromOrigin(100) −> plotPoint(p, red) }
  case { p : DistanceFromOrigin(250) −> plotPoint(p, darkred) }
  case { p : Point −> plotPoint(p, default) }
```

Consequences

—**Benefits.** We can provide whatever destructured data we need to match against, so our matches can say exactly what they require.

—**Liabilities.** We may need to update our custom destructuring pattern when the objects it matches change. We may need many specialised patterns, which might be clearer as conditionals inline where they are used.

Discussion

A custom destructuring pattern is useful when working with code already using match-case, when code will frequently want to match on only the objects containing a particular value, or when concisely obtaining a computed value from the state in a match is useful. For example, the default pattern for a Shape object might extract to the coordinates, but it would also be useful to extract the area at times. Custom Destructuring Pattern allows computing that value and making it available with the same interface as other destructured data.

Related Patterns

—Destructuring Matching describes the use of built-in destructuring patterns.

—All custom destructuring patterns are also Custom Patterns.

Known Uses

Breeze uses Custom Destructuring Pattern heavily, with its numeric and language data structures providing pattern access to their encapsulated values. Minigrace uses Custom Destructuring pattern to implement built-in patterns.

11. LAMBDA PATTERNS

How do you define a pattern that sets what is bound as the matched object?

Sometimes we may want to process the target of the match and create a different object to use in our code, derived from the original target. How can we define a single-use pattern near to where we use it?

Forces

—You want to match against one object, but potentially use a different value as the result.

—You only want to use the pattern in one place.

Therefore: *Use a matching block.*

A matching block, as used in the case clauses of match-case, is itself a pattern with a match method. The return value of the block is used as the value matched against. match-case itself requests the match method on the matching blocks in turn until a SuccessfulMatch is obtained, returning the result of that match.

Example

Suppose we allow querying our sets of data points, returning the set matching some criterion. We can write this method to select all items matching a given pattern:

```
method filter(data : Set)by(pat : Pattern) {
  def values = set.new // To accumulate selected values.
  for (data) do { v ->
    def m = pat.match(v)
    if (m) then {
      values.add(m.result)
    }
  }
  values
}
```

If the data we are really interested in is not the data points themselves but rather some derived aspect like the distance from the origin, we can use a lambda pattern to match and derive in one fell swoop:

```
def rightDists = filterProject(currentData)by { p : Point(x : Positive, y) -> ((x^2) + (y^2)).sqrt }
```

Our lambda pattern matches any Point with a positive x value, but gives as a result not the point itself but the calculated distance from $(0, 0)$.

Consequences

—**Benefits.** We can perform matching and processing in one step. We do not need to write a custom pattern to perform complex matching.

—**Liabilities.** A matching block as a pattern can be difficult to follow.

Discussion

A lambda pattern is most useful with methods behaving similarly to match-case itself, or like map or filter operations. For simply computing the result of a match, Custom Pattern will usually be more understandable. Methods that could naturally take a single-argument block as argument and apply it repeatedly often adapt easily to taking a pattern instead without loss of generality but with enhanced functionality. Library code should consider accepting lambda patterns where suitable. User code should use a lambda pattern when the library method supports it and they want to restrict the results or destructure the target objects somehow.

Related Patterns

—The Match Method is how a lambda pattern's matching is used.

—Custom Pattern is a more powerful still way of customising the behaviour of matching.

—Any pattern using match-case uses lambda patterns for each case clause.

Known Uses

Minigrace uses Lambda Pattern in the implementation of match()case() itself, as well as to implement a combined filter-map operation over the abstract syntax tree.

12. GENERAL KNOWN USES

Minigrace, a compiler for Grace, uses all of these patterns other than Custom Destructuring Pattern. Pattern-matching is very useful for processing the abstract syntax tree. The AST at many points contains nodes of a variety of concrete types, some of which must be handled differently in context.

```
match (node)
  case { t : ASTType −> compileType(t) }
  case { i : Identifier("self") −> compileSelfReference }
  case { i : Identifier −> compileIdentifier(i) }
  case { l : ASTNumber | ASTString | ASTBoolean −> compileLiteral(l) }
  case { _ −> util.syntaxError "invalid argument in context" }
```

FlockDB [FlockDB], a distributed graph database written in Scala [Scala], uses several of these patterns with Scala's pattern-matching feature. Match Method is the fundamental interface to pattern-matching for it, while Literal Matching, Wildcard Pattern, and Type Matching are used to differentiate different values in its data:

```
private def opposite(direction: String) = direction match {
  case "ASC" => "DESC"
  case "DESC" => "ASC"
  case "<" => ">="
  case ">" => "<="
}
def toInt = {
  wrapped match {
    case i: Int => i
    case n: Long => n.toInt
  }
}
override def equals(that: Any) = that match {
  case that: ResultWindow[_] => iterator.toList == that.iterator.toList && nextCursor == that.nextCursor &&
    prevCursor == that.prevCursor && cursor == that.cursor
  case _ => false
}
```

The patterns produce shorter and clearer code than the alternative implementations.

Breeze, a library for numeric processing and AI written in Scala, uses all of the patterns described here. Destructuring Matching is used extensively and the data structures implement Custom Destructuring Pattern. A user of the library can match against or extract the values encapsulated in the numerical or language objects without concern for the particular interface or kind of object involved.

Appendix: Grace in a Nutshell

This appendix gives an overview of the Grace language, and is derived from a section of Black et al. [2012].

Grace is an imperative object-oriented language with block structure, single dispatch, and many familiar features. Our design choices have been guided by the desire to make Grace look as familiar as possible to instructors who know other object-oriented languages, and by the need to give instructors and text-book authors the freedom to choose their own teaching sequence. Thus, in Grace it is possible to start using types from the first, or to introduce them later, or not at all. It is also possible to start with objects, or with classes, or with functions. Most importantly, instructors can move from one approach to another while staying within the same language.

Grace can be regarded as either a class-based or an object-based language, with single inheritance. A Grace class is an object with a single factory method that returns an object:

```
class aCat.named(n : String) {
    def name = n
    method meow { print "Meow" }
}
var theFirstCat:Cat := aCat.named "Timothy"
```

Here the class is called aCat and the factory method named(). After executing this code sequence, theFirstCat is bound to an object with two attributes: a constant field (name), and a method meow. The expression c.name answers the string object `"Timothy"` and c.meow has the effect of printing *Meow*.

An object can also be constructed using an object literal — a particular form of Grace expression that creates a new object when it is executed. In addition to fields and methods, an object literal can also contain code, which is executed when the object literal is evaluated. For example:

```
var theSecondCat := object {
    def name = "Timothy"
    method meow { print "Meow" }
    print "Timothy now exists!"
}
```

This code has the effect of printing “Timothy now exists!”, and binding the variable theSecondCat to a newly-created object, which happens to be operationally equivalent to theFirstCat.

A class is equivalent to an object with a factory method that contains an object literal. Thus, an instructor who wishes to start teaching with objects need not talk about classes at all until later. Classes are in the language because we felt that they were important for convenience, and to help make the connection between Grace and existing languages.

Mutable and immutable bindings are distinguished by keyword: **var** defines a name with a variable binding, which can be changed using the := operator, whereas **def** defines a constant binding, initialized using =, as shown here.

```
var currentWord := "hello"
def world = "world"
...
currentWord := "new"
```

The keywords **var** and **def** are used to declare both local bindings and fields inside objects.

An object’s methods are immutable, in the sense that once an object is created, the code of its methods cannot be changed. A field that is declared with **def** is constant; the binding between the field name and the object cannot be changed, although the object, if mutable, may change its state. Each constant field declaration creates an accessor method on the object. For example, the object club defined by

```
def club = object {
    def members = MutableSet.empty
}
```

has a *method* called members that returns the current set of members. The value of this set may change over time, for example, after executing club.members.add(anApplicant).

Declaring a field with **var** creates two accessor methods, one for fetching the currently bound object and one for changing it. So, after the declaration

```
def car = object {
    def numberOfSeats = 4
    var speed: Number := 0.
}
```

the object car will have three methods called numberOfSeats, speed, and speed:=(). When we use () in the name of a method, it indicates the need to supply arguments. So, the last method might be used by writing car.speed := 30. Variable fields have no value until they are initialized. Because Grace does not define a universal "nil" object, there is no default value that could be used to initialize all variables. We expect that most variable fields will be initialized when they are declared, as in the example above. Attempting to access an uninitialized field is an error that the implementation will detect, and that will cause program termination.

Grace will support visibility annotations that allow the programmer to restrict access to fields and methods from outside an object by marking them as *public* or *confidential*. For simplicity, we do not discuss this further here, and omit visibility annotations in all the sample code in this paper.

In Grace we say that a method is invoked using a "method request". We introduce this terminology to distinguish the operation — fundamental to object-orientation — of *asking* an object to do something, where the choice of what to do is made by the object itself, from procedure or function call, where the choice of operation is made by "the caller". This distinction is also conveyed by Smalltalk's "message send" terminology, but now that networks and distributed systems are ubiquitous, "sending a message" has become an ambiguous term.

All the attributes of an object (methods, variable fields, and constant fields) exist in the same namespace and with the same lookup rules. There is thus a potential ambiguity in the interpretation of a name n. To resolve this, we disallow "shadowing" of variable names in enclosing scopes; one or other of the variables must be renamed. Thus, if n is defined in the local scope, it cannot also be imported from an enclosing scope. It may be inherited, in which case the local definition overrides the inherited one. If there is no local declaration of n, you might wonder whether n is a reference to a variable (or constant) declared in an enclosing (static) scope, or a request **self**.n of an inherited method. To remove this ambiguity, we disallow this situation too. We do not allow n to be declared in a statically enclosing scope if it is also inherited. If that is the case, the programmer must rename the variable n in the enclosing scope.

Our choices might be too restrictive for an industrial-strength programming language, but for teaching purposes we believe them to be appropriate.

Making field access syntactically identical to a self method request is deliberate, following the lead of Eiffel and Self. This allows the programmer to override a field's implicitly-defined accessor method with a custom method, and thus allows the programmer to handle interface changes, implement bounds checking, or perform some other similar task without affecting the interface of the object.

An object containing no mutable state is by default equal to any other object with the same structure and values for its fields. This follows Baker's Egal predicate. The equality method that implements this test is generated automatically.

Grace method names may consist of multiple parts ("mixfix" notation) as in Smalltalk. Separate lists of arguments are interleaved between the parts of the name, allowing them to be clearly labelled with their purpose. Thus we might define on Number objects

```
method between (l:Number) and (u:Number) {
    return (l < self) && (self < u)
}
```

The syntax of a method request is similar to that used in Java, C++, and many other object-oriented languages: obj.meth(arg1, arg2), but extended to allow meth to have multiple parts. We could request the above method between()and() on 7 by writing

```
7.between(5) and(9)
```

Single arguments that are literals do not require parentheses, so alternatively we could write

```
7.between 5 and 9
```

Following many other languages, the receiver **self** can be omitted. We have already seen several messages requested of an implicit receiver; for example, print `"Meow"` is short for **self**.print `"Meow"`.

Grace also allows operator symbols (and sequences of operator symbols) to be used to name methods. A method name composed of operator symbols is used as a binary infix operator, unless it is defined using the **prefix** keyword, in which case it is a unary prefix operator. There is no ambiguity because the receiver must be explicit when requesting operator methods. Thus, a − b is a request of the binary minus method on object a, while − b is a request of the prefix negation method on object b.

Grace includes first-class blocks (lambda expressions). A block is written between braces and contains some piece of code for deferred execution. A block may have arguments, which are separated from the code by −>, so the successor function is {x −> 1+x}. A block can refer to names bound in its surrounding lexical scope, and returns the value of the last-evaluated expression in its body.

Control structures in Grace are methods. The built-in structures are defined in the basic library, but an instructor or library designer may replace or add to them. Control structures are designed to look familiar to users of other languages:

```
if (x > 5) then {
    print "Greater than five"
} else {
    print "Too small"
}

for (node.children) do { child −>
    process(child)
}
```

Notice that the use of braces and parentheses is not arbitrary: parenthesized expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may thus be evaluated zero, one, or many times. A **return** statement inside a block terminates the *method* that lexically encloses the block, so it is possible to program *quick exits* from a method by returning from the *then* block of an if()then() or the *do* block of a while()do().

Instructors could provide their own for()do() methods with debugging enhancements or additional restrictions, either to replace the built-in version or to sit alongside it. A particular use of this facility is to define iterators that require the student to specify loop invariants.

String literals, written between double quotes, support interpolation, using a syntax similar to that of Ruby. Code inside braces within a literal is evaluated when the string is; the asString method is requested on the resulting object, and the answer is inserted into the string literal at that point.

```
print "1 + 2 = {1 + 2}" // Prints "1 + 2 = 3"
```

While Grace uses braces to delimit blocks and other literals, it also enforces correct indentation. Braces and indentation may not be inconsistent with one another: the body of a method, for example, must be indented. Enforcing this in the language ensures that students will learn good practice, and avoids the common problem of not being able to find a mismatched brace because of the tendency of one's eye to believe the indentation rather than the braces.

Grace code can also be written in "script" form, without object or class definitions. We imagine such code as being enclosed in a top-level object literal. Thus, methods can be defined at the top level, and any code written at the top level will be executed immediately. This supports imperative-first teaching styles.

REFERENCES

BLACK, A. P., BRUCE, K. B., HOMER, M., AND NOBLE, J. 2012. Grace: the absence of (inessential) difficulty. Accessible from `http://gracelang.org/documents/`.

FlockDB. FlockDB web site. `https://github.com/twitter/flockdb`.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Grace. The Grace programming language. `http://gracelang.org/`.

Scala. Scala web site. `http://www.scala-lang.org/`.

WIRTH, N. AND HOARE, C. A. R. 1966. A contribution to the development of algol. *Commun. ACM 9,* 6, 413–432.