The Grace Programming Language
Draft Specification Version 0.095

Andrew P. Black Kim B. Bruce James Noble

February 22, 2011

1 Introduction

This is a specification of the Grace Programming Language. This specification is notably incomplete, and everything is subject to change. In particular, this version does *not* address:

- whether to support object nesting (Beta, Scala, Newspeak) or not (Smalltalk, Python).
- static type system
- encapsulation system
- module system
- collection syntax and collection literals
- annotations
- purity and non-nulls.
- reflection
- assertions, data-structure invariants, pre & post conditions, contracts
- regexps
- libraries

For discussion and rationale, see `http://gracelang.org`.

Where this document gives "(options)", we outline choices in the language design that have yet to be made.

2 User Model

> *All designers in fact have user and use models consciously or subconsciously in mind as they work. Team design... requires explicit models and assumptions.*
>
> Frederick P. Brooks, *The Design of Design*. 2010.

1. First year university students learning programming in CS1 and CS2 classes that are based on object-oriented programming.
   (a) The courses may be structured objects first, or imperative first. Is it necessary to support "procedures first"?
   (b) The courses may be taught using dynamic types, static types, or both in combination (in either order).
   (c) We aim to offer some (but not necessarily complete) support for "functional first" curricula, primarily for courses that proceed rapidly to imperative and object-oriented programming.
2. University students taking second year classes in programming; algorithms and data structures, concurrent programming, software craft, and software design.
3. Faculty and teaching assistants developing libraries, frameworks, examples, problems and solutions, for first and second year programming classes.
4. Programming language researchers needing a contemporary object-oriented programming language as a research vehicle.
5. Designers of other programming or scripting languages in search of a good example of contemporary OO language design.

3 Syntax

> *Much of the following text assumes the reader has a minimal grasp of computer terminology and a "feeling" for the structure of a program.*
>
> Kathleen Jensen and Niklaus Wirth, *Pascal: User Manual and Report.*

Grace programs are written in UTF-8. Reserved words and the names of library methods are written in the ASCII subset of UTF-8.

3.1 Layout

Grace uses curly brackets for grouping, and semicolons as statement terminators. Grace also uses indentation (layout) for grouping, and infers semicolons at the end of lines. If code uses both layout and braces, then it's a syntax error for them not to agree.

code with punctuation:

```
while {stream.hasNext} do {
    print(stream.read);
};
```

code with layout:

```
while
      stream.hasNext
   do
      print(stream.read)
```

code with both:

```
while {stream.hasNext} do
    print(stream.read)
```

3.2 Comments

Grace's comments delimiters follow C++ and Java. However, comments are *not* treated as white-space; each comment is conceptually attached to the smallest immediately preceeding syntactic unit.

```
// comment to end-of-line
/* comment to close comment */
```

3.3 Identifiers

Identifiers in Grace must begin with a letter and consist of letters and digits thereafter.

(option) Prime ' and underscore characters _ may be used after the first character of an identifier.

3.4 Reserved Words

The ? indicates words related to design options not yet chosen.

self super? true false method const var next? outer? prefix? constructor? object class extends type

3.5 Tabs and Control Characters

Newline can be represented either by carriage return or by line feed; however, a line feed that immediately follows a carriage return is ignored.

Tabs and all other non-printing control characters (except carriage and line feed) are syntax errors.

4 Built-in Objects

4.1 Numbers

All numeric types are sub-types of an abstract type Number. All Grace implementations must support both exact Rational and IEEE754.Binary64 floating point numbers. Implementations may support other numeric types.

Grace has three syntactic forms for numerals (literals that denote Numbers):

1. decimal numerals, written as strings of digits, optionally preceded by a minus;
2. explicit radix numerals, written as a (decimal) number between 2 and 35 representing the radix, a leading x, and a string of digits, where the digits from 10 to 35 are represented by the letters A to Z, in either upper or lower case. As a special case, a radix of 0 is taken to mean a radix of 16. Explicit radix numerals may *not* be preceded by a minus.
3. base-exponent numerals, always in decimal, which use e as the exponent indicator. Base-exponent numerals may be preceded by a minus.

All literals evaluate to exact Rational numbers; explicit conversions (such as b64) must be used to convert rationals to other types.

Examples

```
1
−1
42
3.14159265
13.343e−12
−414.45e3
16xF00F00
2x10110100
0xdeadbeef // Radix zero treated as 16
```

4.2 Booleans

The leywords **true** and **false** denote the only two values of Grace's Boolean type. Boolean operators will generally be written using single characters & for and, | for or.

"Short circuit" (a.k.a non-commutative) boolean operators use keyword messages that take blocks as their second arguments.

Examples

```
P & Q
toBe | toBe.not
container.nonEmpty.and {container.first = sought}
```

4.3 Strings and Characters

String literals in Grace are written between double quotes, as in C, Java, and Python. Strings literals support a range of escape characters such as "\t\b\n\f\r\v\\\"", and also escapes for Unicode. Individual characters are represented by Strings of length 1. Strings are immutable Grace values (see §10) and so may be interned. Strings conform to the protocol of an immutable IndexableCollection, and Grace's standard library will include mechanisms to support efficient incremental string construction.

Examples

```
"Hello World!"
"\t"
"The End of the Line\n"
"A"
```

4.4 (option) String interpolation

We are considering syntax so that strings (or expressions returning objects that support the asString method) can be directly interpolated into strings.

Examples

```
"Adding {a} to {b} gives {a+b}"
```

5 Blocks

Grace blocks are parameterless lambda expressions. Blocks with parameters have a formal parameter list terminated by the => symbol.

Blocks are evaluated by sending them the message apply[1]

```
{ print "Hello World" }

if (x = 3)
    print "Three!"

for 1..10 do
    i => print i


var sum := 0
const summingBlock : Block<Number,Number> =
    { i:Number => sum := sum + i }
summingBlock.apply(4)          // sum is now 4
summingBlock.apply(32)         // sum ins now 36
```

Blocks are lexically scoped inside their containing method, or block. A "naked" block literal that is neither the target of a message send nor passed as an argument is a syntax error.

The body of a block consists of a sequence of declarations and expressions (option) and also statements, if we have them.

[1] Isn't apply overloaded and variadic?

6 Declarations

Const and var declarations may occur anywhere within a method or block: their scope is to the end of their defining block or method.

6.1 Constants

Constants bind an identifier to the value of an initializer expression, optionally at a precise type.

Examples

```
const x = 3 * 100 * 0.01
const x:Rational = 3 // probably means the same as the above
const x:Binary64 = 3
const x:Rational // Syntax Error: x must be initialised
```

Grace has a single namespace for methods and constants (and variables). A constant declaration of x can be seen as creating a (nullary) accessor method x.

6.2 Variables

Grace supports variable declarations using the "**var**" keyword.

Uninitialized variables (of any type) are given a special "uninitialized" value; accessing this value is an error (caught either at run time or at compile time, depending on the cleverness of your implementor).

Examples

```
var x := 3              // type of x is inferred.
var x:Rational := 3     // explicit type.
var x:Binary64 := 3     // explicit type
```

Variables are reassigned using assignment methods (see §8.2). A variable declaration of "x" can be seen as creating an accessor method "x" and an assignment method "x:=" Grace's encapsulation system will control the accessibility of each of these methods.

6.3 Methods

Methods are declared with the "**method**" keyword, a name, optionally an argument list, potentially repeated, optionally a return type declaration,

and a method body. Where a method body is a single expression, it may be introduced by an "=" and the body needs no "**return**" statement.

Assignment methods are named by an identifier suffixed with ":=". If supported, prefix operator methods will be named "**prefix**" followed by the operator character(s).

Examples

```
method pi = 3.141592634

method greetUser {print ''Hello World!''}

method +(other : Point) : Point = { (x+other.x) @ (y+other.y) }

method + other
        (x+other.x) @ (y+other.y)

method +(other)
        return (x+other.x) @ (y+other.y)

method foo:=(n : Number) returns Void
        print "Foo currently {foo}, now assigned {n}"
        super.foo:= n

method choseBetween (a : Block<Void>) and (b : Block<Void>) returns Void
    if Random.nextBoolean
        then {a.apply} else {b.apply}

class Number {
    method prefix− returns Number = {0 − self}
}
```

7 Objects and Classes

Grace **object** constructor expressions and declarations produce individual objects. Grace provides **class** declarations to create classes of objects all with the same structure.

Grace's class and inheritance design is not yet complete. In particular, details of multiple factory methods, and factory methods across inheritance hierarchies are likely to change.

7.1 Objects

Objects are created by primitive object constructor literals. The body of an object constructor literal consists of a sequence of declarations.

```
object
    const color : Color = Color.tabby
    const name : String = "Unnamed"
    var miceEaten := 0
```

Objects constructors are lexically scoped inside their containing method, or block. In particular, any initializer expressions on fields or constants are executed in that lexical context. (The durability of that link is the "nesting" question, see §1). Each time an object constructor is executed, a new object is created.

A constant initialized by an object constructor, such as

```
const unnamedCat = object
    const color : Color = Color.tabby
    const name : String = "Unnamed"
    var miceEaten := 0
```

evaluates that constructor just once, and repeated invocations of the reader method unnamedCat will return the same object. This may be abbreviated with a named object declaration that creates a singleton object.

```
object unnamedCat
    const color : Color = Color.tabby
    const name : String = "Unnamed"
    var miceEaten := 0
```

7.2 Classes

Objects declarations have no provision for initialising objects other than lexical scope. Class declarations combine the definition of an object with the definition of a factory object with a method that creates instances of the class. A class declaration is similar to a named object declaration, except that it may have one or more methods annotated <factory>:

Examples

```
class Cat
    const color : Color
```

```
const name : String
var miceEaten := 0
<factory> newUnnamedTabby() returns Cat
    return new(Color.tabby, "Unnamed")
<primary factory> new(color : Colour, name : String) returns Cat
```

A *primary* factory method has no body; it creates a new object, with its constants and variables inititalized from the method's lexical scope. That is, in the above example, the constants color and name are initialised from the parameters of new with the same names. Other factory methods may create objects by calling the primary factory method.

The new object that is created is the object that one gets by:

1. replacing the keyword **class** with **object**,
2. and deleting the factory attributes.

New instances of the class — new objects of the above shape — are created in the program by sending one of the factory messages to the factory object.

Examples

```
const stray = Cat.newUnnamedTabby // create new unnamed cat
const fergus = Cat.new("tortoiseshell", "Fergus Trouble")
```

7.3 Inheritance

Grace class declarations supports inheritance with "single subclassing, multiple subtyping" as in Java. A new declaration of a method or field can override an existing definition — overriding declarations must be declared <override>, and overridden defintions can be accessed via **super** calls §8.5. It is a static error for a field to override another field or a method. This example shows how a subclass can override accessor methods for a variable defined in a superclass (in this case, to always return 0 and to ignore assignments).

```
class PedigreeCat extends Cat implements CatShowExhibit
    var prizes = 0
    <override> method miceEaten = 0;
    <override> method miceEaten:=(n :Number) {return} //Just forget
    <factory> new(color : Colour, name : String) : Cat {}
```

Note that a subclasses' factory method should initialize not just its own uninitialized fields, but all uninitialized fields declared in its superclasses. (see 6.2)

8 Method Calls

Grace is a pure object-oriented language. Everything in the language is an object, and all computation proceeds by calling dynamically dispatched methods on objects.

8.1 Named Method

A named method call is a receiver followed by a dot ".", then a method name (an identifier), then any arguments in parentheses. If there are no arguments, or only one argument, the parentheses may be omitted. A long argument list may be divided by additional "keyword" method names. A named method call with an implicit receiver omits the receiver and the dot.

```
graphicsContext.drawLineFrom(source) to( destination )
graphicsContext.movePenTo(x,y)

print ("Hello world")
print "Hello world"

pt.x()
pt.x
```

Grace does not allow overloading on argument type. Grace (currently) does not support variadic methods.

8.2 Assignment Method

A assignment method call is an explicit receiver followed by a dot, then a message name (an identifier) directly followed by "`:=`", and then a single argument. An assignment method call with an implicit receiver omits the receiver and the dot.

```
x := 3
y:=2
widget.active := true
```

8.3 Binary Operator Method

Grace also allows operator symbols (sequences of operator characters) for binary messages — with an explicit receiver and one argument. An operator method name is one or more operator characters, and may not match a reserved symbol (for example "." is reserved, but ".." is not). All Grace operators have the same precedence: it is a syntax error for two different operator symbols to appear in an expression without parenthesis to indicate order or evaluation. The same operator symbol can be sent more than once without parenthesis and is evaluated left-to-right.

```
1 + 2 + 3 // evaluates to 6
1 + (2 * 3) // evaluates to 7
(1 + 2) * 3 // evaluates to 9
1 + 2 * 3 // syntax error in Grace; would evaluate to 9 in Smalltalk
```

Named message calls bind more tightly than operator message calls: The following examples show first the Grace expressions as they would be written, followed by the parse

```
1 + 2.i                                  1 + (2.i)
(a * a) + (b * b).sqrt                   (a * a) + ((b *b).sqrt)
((a * a) + (b * b)).sqrt                 ((a * a) + (b *b)).sqrt
a * a + b * b                        // syntax error
a + b + c                                (a + b) + c
a     b     c                                (a     b)     c
```

8.4 (option)Unary Prefix Operator Method

We are considering permitting unary prefix operator methods — since we do not support binary operator methods with implicit receivers there is no syntactic ambiguity.

```
− (b +/− (4 * a * c).sqrt)

status.ok := ! engire_fire  & !wings_attached & on_course
```

Prefix operators would bind tighter than binary operators or other method calls.

8.5 Super calls

The reserved word **super** may only be used as an explicit receiver. A "super" call invokes the method that would have been called had the currently

executing method not been defined.

```
super.foo
super.bar(1,2,6)
super.doThis(3) timesTo("foo")
super + 1
!super

foo(super)  // syntax error
1 + super   // syntax error
```

8.6 Encapsulation

The design of Grace's encapuslation system has not yet begun in earnest.

Grace will support some kind of private methods that can only be called on **self** or **super**.

9 Control Flow

Control flow statements in Grace are syntactically message calls. While the design of the module system is not complete (not yet begun) we expect that instructors will need to define domain-specific control flow constructs in libraries — and these constructs should look the same as the rest of Grace.

9.1 Basic Control Flow

If statements:

```
if (test) then {block}

if (test) then {block} else {block}
```

While statement:

```
while {test} do {block}
```

For statement:

```
for ( collection ) do {each => block}

for (course.students) do { each => print each }

for 0..n do { i => print i }
```

9.2 Case

Grace will support a match/case construct:

```
match (x)
// match against a literal constant
  case { 0 => "Zero" }

// typematch, binding a variable ''' looks like normal parameter passing case '''
  case { s : String => print(s) }

// match against an existing name – requiring parens like Scala
  case { (pi) => print("Pi = " ++ pi) }

// destructuring match, binding variables ...
  case { Some(v) => print(v) }
```

This is the "pattern-matching-lambda" design (see blog post 10/02/2011).

9.3 Exceptions

Grace supports basic unchecked exceptions. Exceptions will be generated by the "raise" keyword with an argument of some subtype of Exception:

```
  raise UserException.new("Oops...!")
```

Exceptions are caught by a "catch / case" statement that syntactically parallels the "match / case" statement.

```
catch {File.open("data.store")}
  case {e : NoSuchFile => print("No Such File"); return}
  case {e : PermissionError => print("No Such File"); return}
  case {Exception => print("Unidentified Error"); System.exit()}
```

Exceptions can't be restarted, however, the stack frames that are terminated when an exception is raised should be pickled so that they can be used in the error reporting machinery (debugger, stack trace).

10 Equality and Value Objects

All objects will automatically implement the following methods, and programmers are not able to override them.

1. "=" and "!=" operators implemented as per Henry Baker's EGAL predicate.

2. "hashcode" compatible with the EGAL operators.

As a consequence, immutable objects (no "**var**" fields, and only containing other immutable objects) will act as pure "value objects". A Grace implementation can support value objects using whatever implementation is most efficient: either passing by reference always, by passing some types by value, or even by inlining fields into their containing objects, and doing a field-update if the containing object assigns in a new value.

11 Types

The design of Grace's type system has not begun in earnest.

We expect to provide type declarations describing objects' interfaces.

Named object or class declarations will define types with the same name as object or class begin defined. The various Cat object and class descriptions (see §7) would produce a type such as:

```
type Cat
    color : Color
    name : String
    miceEaten : Number
    miceEaten:=(Number) : Void
```

1. Classes are not types.
2. Classes define a type (of the same name)
3. Grace will probably use structural subtyping

Acknowledgements

The Scala language specification 2.8 and the Newspeak language specification 0.05 were used as references for early versions of this document. Please excuse the lack of citations so far in this draft.

A To Be Done

As well as the large list in Section 1 of features we haven't started to design, this section lists details of the language that remain to be done:

1. specify full layout syntax §3.1
2. decide about variadic methods §8.1 including Block::apply §5
3. character to separate block args from body => (Scala, C♯), -> (Java8, ML, Haskell), | (Smalltalk, Self, Newspeak) §5
4. consider whether to follow Self message name syntax (second and subsequent keyword must begin with a Capital) §8.1
5. decide about prefix operators §8.4
6. short-circuit boolean operators §4.2
7. string interpolation syntax §4.4
8. confirm operator precedence §8
9. confirm method lookup algorithm, in particular relation between lexical scope and inheritance 8
10. make the **const** keyword optional, or remove it §6.1
11. how powerful should **case** be §9.2, see blog post 10/02/2011.
12. still unhappy with inheritance design §7.3
13. where should we draw the lines between object constructor expressions/named object declarations, class declarations, and "hand-built" classes? §7.3
14. how do factories etc relate to "uninitialized" §6.2
15. what syntax for control structure "arguments" parentheses or braces? §9.1
16. James is tempted by Ruby style exceptions — where handlers can be nested at the end of any block. §9.3
17. What is the namespace of types? What is the syntax of types? §11 More to the point, what is the type system?