The Grace Programming Language
Draft Specification Version 0.132

Andrew P. Black Kim B. Bruce James Noble

October 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).
- collection syntax and collection literals
- tuples vs multiple values vs multiple returns
- nested static type system (although we've made a start)
- encapsulation system
- module system
- metadata (Java's @annotations, C♯ attributes, final, abstract etc)
- purity and non-nulls.
- reflection
- assertions, data-structure invariants, pre & post conditions, contracts
- regexps
- test support
- libraries, including more Numeric types

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 Unicode. Reserved words are written in the ASCII subset of Unicode. As a matter of policy, the names of methods defined in the required libraries are also restricted to the ASCII subset of Unicode.

3.1 Layout

Grace uses curly brackets for grouping, and semicolons as statement terminators, and infers semicolons at the end of lines. Code layout cannot be inconsistent with grouping.

code with punctuation:

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

code without punctuation:

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

A line break followed by an increase in the indent level implies a line continuation, whereas line break followed by the next line at the same or lesser indentation implies a semicolon (if one is permitted syntatically).

3.2 Comments

Grace's comments delimiters follow C++ and Java's line ("//") comments. Comments are *not* treated as white-space; each comment is conceptually attached to the smallest immediately preceding syntactic unit; comments following a blank line are attached to the largest immediately following syntactic unit.

```
// comment to end−of−line
```

3.3 Identifiers

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

Prime ' characters may be used after the first character of an identifier.

An underscore "_" acts as a placeholder identifier: it is treated as a fresh identifier everywhere it is used.

3.4 Reserved Words

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

> **assert case catch class const def extends false finally match method object outer**(?) **prefix raise return self super true type var where**

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, even in a string literal. (There are escape sequences for including special characters in string literals.)

4 Built-in Objects

4.1 Numbers

Grace will support a single type Number. Number will maintain rational computations in arbitrary precision, and inexact irrational computations approximated to at least 64bit precision.

Implementations may support other numeric types: a full specification of numeric types is yet to be completed.

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 f64) 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 keywords **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, and prefix ! for not.

Examples

```
P & Q
toBe | toBe.not
```

"Short circuit" (a.k.a non-commutative) boolean operators take blocks as their second argument:
Examples

```
P && { Q }
toBe || { !toBe }
```

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", and also escapes for Unicode; these are listed in Table 1 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.

Escape	Meaning	Escape	Meaning
\\	backslash	\'	single quote
\"	double quote	\b	backspace
\n	line-feed	\r	carriage-return
\t	tab	\l	unicode newline
\f	page down	\e	escape.
\{	left bracket	\}	right bracket
\ (\ space)	non-breaking space		

Table 1: Grace string escapes. A platform-dependent newline is either a line-feed (lf) or a carriage-return (cr) or a cr-lf pair, depending on the platform.

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 lambda expressions; they may or may not have parameters. If a parameter list is present, the parameters are separated by commas and terminated by the −> symbol.

```
{do.something}
{ i -> i+1}
{ sum, next -> sum +next }
```

Blocks construct objects with a single method named apply, or apply(n) if the block has parameters. The block is evaluated by requesting the apply method with the same number of arguments as the block has parameters. It's an error to provide fewer or more parameters.

```
for (1..10) do {
    i -> print i
}
```

might be implemented as

```
method for (collection) do (block) {
        ...
        block.apply( collection .at(i))
        ...
}
```

Here is another example:

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

Blocks are lexically scoped inside their containing method or block. A "naked" block literal that is neither the target of a method request 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.

6 Declarations

Def and var declarations may occur anywhere within a method or block: their scope is the whole of their defining block or method.

It is an error to declare an identifier that shadows a lexically enclosing identifier.

6.1 Constants

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

Examples

```
def x = 3 * 100 * 0.01
def x:Number =3          // means the same as the above
def x:Number    // 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) reader 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.
```

Instance variables are reassigned using assignment methods (see §8.2). A variable declaration of “x” can be seen as creating a reader method “x” and an assignment method “x:=(_)” Grace’s encapsulation system will control the accessibility of each of these methods. You can think of the real instance variable as having a unique secret name, which is known only to the accessor methods.

Block and method temporary variables really exist, and can be the targets of real assignment statements.

It’s a deliberate design decision that assignment to a local variable and requesting an assignment method on an object look identical.

It is an error to declare a block or method temporary variable that shadows an enclosing method or assignment method (see §6)

Assignments return Nothing (Void/None/etc).

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. Methods may not be nested.

Methods may contain one or more **return** e statements. If a **return** statement is executed, the method terminates with the value of the expression e. If the method **returns** None, then no expression may follow the **return**. If execution reaches the end of the method body without executing a **return**, the method terminates and returns the value of the last expression evaluated.

Assignment methods are named by an identifier suffixed with ":=".

Prefix operator methods will be named "**prefix**" followed by the operator character(s).

Methods may have a "repeated parameter" to provide variable arity (aka varargs). A repeated parameter, if present, must be the last parameter to a method, and must be the only parameter after the last part of a multi-part method name. Repeated parameters are designated by a prefix star (asterix, "*") before the name of the parameter. Inside the method, the repeated parameter has the type of an immutable collection of the declared type — a parameter declared foo(*args : String) will have a type such as args : ImmutableCollection<String>

Methods may optionally be declared or requested with generic type parameters. Formal generic type parametrs may be constriained with **where** clauses.

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) -> None {
```

```
            print "Foo currently {foo}, now assigned {n}"
            super.foo:= n }

    method choseBetween (a : Block<None>) and (b : Block<None>) −> None {
        if (Random.nextBoolean)
            then {a.apply} else {b.apply} }

    method print( ∗args : Printable ) −> None

    method sumSq<T>(a : T, b : T) −> T where T <: Numeric {(a ∗a) +(b ∗
    b)}

    class NumberFactory {
        method prefix− −> 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 of which have the same structure.

Grace's class and inheritance design is complete but tentative. We need experience before confirming the design.

7.1 Objects

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

```
    object {
        def colour:Colour = Colour.tabby
        def name:String = "Unnamed"
        var miceEaten := 0
    }
```

Object literals are lexically scoped inside their containing method, or block. In particular, any initializer expressions on fields or constants are executed in that lexical context. (Whether methods are also in that scope is the "nesting" question, see §1. The current design is that initializers are nested, but not methods). Each time an object literal is executed, a new object is created.

A constant can be defined by an object literal, such as:

```
def unnamedCat = object {
    def colour : Colour = Colour.tabby
    def name : String = "Unnamed"
    var miceEaten := 0 }
```

to bind a name to an object. Repeated invocations of the reader method unnamedCat will return the same object.

7.2 Classes

Objects literals have no provision for initializing the constant and variable attributes of the created object other than via lexical scope.

Class declarations combine the definition of an object with the definition of a factory object, where the factory object has a method, named new, that creates "instances of the class". A class declaration is similar to an object literal, except that it may have parameters, like a block:

Examples

```
class CatFactory { aColour, aName ->
    def colour : Colour = aColour
    def name : String = aName
    var miceEaten := 0
}
```

The new method takes as many arguments as the class has parameters. The object that is returned by an execution of new has the fields and methods listed in the body of the constructor that follows the **class** keyword. If there are formal parameters to the class body, they are initialized to the arguments to new, and are also in scope within the class.

So, in the above example, the constants colour and name are initialized from the parameters aColour and aName, which are in turn initialized from the first and second arguments to new:

```
def fergus = CatFactory.new(" tortoiseshell ", "Fergus Trouble")
```

If the programmer wants a factory object with more methods, or method names other than new, she is free to build such an object using nested object constructors. The above declaration for **class** Cat is equivalent (modulo types and modules) to the following nested object declarations:

```
def CatFactory = object { // the cat factory
    method new(aColour: Colour, aName: String) −> Cat {
        object { // the cat herself
            def colour : Colour := aColour
            def name : String := aName
            var miceEaten := 0
        }
    }
}
```

Notice that the type Cat describes the object returned from Cat.new, not the factory object CatFactory.

7.3 Inheritance

Grace class declarations supports inheritance with "single subclassing, multiple subtyping" (like Java), by way of an **inherits** C clause in a class declaration or object literal.

A new declaration of a method can override an existing declaration, but overriding declarations must be annotated with <override>. Overridden methods can be accessed via **super** calls §8.6. 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 PedigreeCatFactory { aColour, aName −>
    inherits Cat.new(aColour, aName)
    var prizes := 0
    <override> method miceEaten {0};
    <override> method miceEaten:= (n:Number) {return} //Just ignore
}
```

The right hand side of an **inherits** clause is restricted to be a class name, followed by a correct request for that class's new method.

7.4 Understanding Inheritance (under discussion)

Grace's class declarations can be understood in terms of a flattening translation to object constructor expressions that build the factory object. Understanding this translation lets expert programmers build more flexible factories.

The above declaration for **class** PedigreeCat is broadly equivalent to the following nested object declarations, not considering types, modules, and *renaming superclass methods so that an object's method names must actually be unique.*

```
def PedigreeCatFactory = object { // the cat factory
    method new(aColour: Colour, aName: String) -> PedigreeCat {
        object { // the cat herself
            def colour : Colour := aColour
            def name : String := aName
            <<private>> var Cat__miceEaten := 0 // ugly. super-ugly
            var prizes = 0
            method miceEaten =0;
            method miceEaten:=(n:Number) {return} //Just ignore
        } // object
     } // method new
} // object
```

7.5 Generic Classes

Classes may optionally be declared or instantiated with generic type parameters. Formal generic type parametrs may be constriained with **where** clauses.

Examples

```
class VectorFactory<T> { size ->
   var contents := Array.size(size)
   method at(index : Number) -> T {return contents.at() }
   method at(index : Number) put(elem : T) { }
}

class SortedVectorFactory<T>
   where T <: Comparable<T> {
 ...
}
```

8 Method Requests

Grace is a pure object-oriented language. Everything in the language is an object, and all computation proceeds by "requesting" an object to execute a method with a particular name. The response of the object is to execute the method. When speaking of Grace, we distinguish the act of *requesting* a method (which is exactly what Smalltalkers call "sending a message"), and involves only a method *name* and some arguments, and *executing* that method, which involves the code of the method, which is always local to the receiver of the request.

8.1 Named Methods

A named method request is a receiver followed by a dot ".", then a method name (an identifier), then any arguments in parentheses. Parentheses are not used if there are no arguments. To improve readability, a long argument list may be interpolated between the "words" that makes up the method name. This is determined by the declaration of the method. If the receiver is **self** it may be left implicit, *i.e.*, the **self** and the dot may both be omitted.

```
canvas.drawLineFrom(source)to(destination)
canvas.movePenToXY(x,y)
canvas.movePenToPoint(p)

print("Hello world")

pt.x
```

Grace does not allow overloading on argument type.

Parenthesis may be omitted where they would enclose a single argument, provided that argument is a block literal, (option) a string literal, or (option) a square bracket literal (if we allow square bracket collection literals).

8.2 Assignment Methods

A assignment method is an explicit receiver followed by a dot, then a method name (an identifier) followed by ":=", and then a single argument. If the receiver is **self** it may be left implicit, *i.e.*, the **self** and the dot may both be omitted.

Examples

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

Assignment methods must return Nothing.

8.3 Binary Operator Methods

Grace allows operator symbols (sequences of operator characters) for binary methods — methods with an explicit receiver and one argument. A binary operator method is one or more operator characters, and may not match a reserved symbol (for example "." is reserved, but ".." is not).

Most 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 of evaluation. The same operator symbol can be sent more than once without parenthesis and is evaluated left-to-right.

Four simple arithmetic operators do have precedence: / and * over + and −.

Examples

```
1 + 2 + 3 // evaluates to 6
1 + (2 * 3) // evaluates to 7
(1 + 2) * 3 // evaluates to 9
1 + 2 * 3 // evalutes to 7
1 +*+ 4 −*− 4 //syntax error
```

Named method requests without arguments bind more tightly than operator method requests. The following examples show first the Grace expressions as they would be written, followed by the parse.

Examples

```
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                (a * a) + (b *b)
a + b + c                    (a + b) + c
a − b − c                    (a − b) − c
```

8.4 Unary Prefix Operator Method

Grace supports unary prefix operator methods: since Grace does not support binary operator methods with implicit receivers there is no syntactic ambiguity.

Prefix operators bind with the same precedence as method requests with no arguments, and therefore need parenthesis to disambiguate.

Examples

```
− (b + (4 ∗ a). sqrt )
− b.squared                 //  illegal
(−b).squared
−(b.squared)

status .ok :=  !( engine. isOnFire) &  wings.areAttached & isOnCourse
```

8.5 Accessing Operator Method

Grace supports an accessing operator [] .

(option) Grace supports a two-argument accessing operator []: =.

Using these operators:

```
print ( a[3] )         // calls method [] on a with argument 3
a[3] := "Hello"        // calls method [] on a with arguments 3 and "Hello"
```

Note: Somewhere we need to have a list of reserved operators that cannot be used normally.

[] :=

8.6 Super Requests

The reserved word **super** may be used only as an explicit receiver. In overriding methods, method requests with the pseudo-receiver **super** request the prior overridden method with the given name from **self**. Note that no "search" is involved; super-requests can be resolved statically, unlike other method requests.

Examples

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

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

8.7 Encapsulation

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

Grace will use metadata annotations support <<private>> methods that can be requested only from **self** or **super**.

8.8 Generic Method Requests

Methods may optionally be requested with actual generic type arguments given explicitly. Where a method declared with formal generic type parameters is requested in a statically typed context without explicit actual generic type arguments, the actual types arguments will be inferred

Examples

```
sumSq<Integer64>(10.i64, 20.i64)

sumSq(10.i64, 20.i64)
```

9 Control Flow

Control flow statements in Grace are syntactically method calls. While the design of the module system is not complete (in fact, hardly 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 {item −> block body}

for (course.students) do { s:Student −> print s }

for (0..n) do { i −> print i }
```

To allow for conventional syntax with a leading keyword (**if**, **while**, **for**), these methods are treated as if they were implicitly sent to **self**, which implies that all objects must inherit the corresponding method.

9.2 Case (under discussion)

Grace will support a match/case construct. Match will take only one argument and match against a series of blocks introduced by "case". Pattern matching will support destructuring.

Examples

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

// typematch, binding a variable − looks like a block with parameter
  case { s:String −> print(s) }

// match against the value in an existing variable − requiring parenthesis like Scala
  case { (pi) −> print("Pi = " ++ pi) }

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

// match against placeholder, matches anythin
  case { _ −> print("did not match") }
```

9.3 Exceptions (under discussion)

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**(_)" construct that syntactically parallels **match**(_)**case**(_)".

```
catch {def f = 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}
   finally {f.close}
```

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). "**catch**(_)**case**(_) **finally** (_)" construct and a "**do**(_) **finally** (_)" construct support finalization even through exceptions. Following Scala, a "**using**(_)**do**(_)" construct supports resource allocation and deallocation:

```
using (Closable.new) do { stranger -> //bound to the new Closable
    stranger.doSomething
}
// the close method is automatically requested of the
// Closable when the block terminates
```

10 Equality and Value Objects

All objects will automatically implement the following non-overridable methods. (option) Library programmers are able to override these methods.

1. == and != operators implemented as per Henry Baker's "egal" predicate [2]. That is, immutable objects are egal if they are of the same "shape" and if their fields' contents are egal, while mutable objects are only ever egal to themselves.

2. hashcode compatible with the egal.

As a consequence, immutable objects (objects with no **var** fields, which capture only other immutable objects) will act as pure "value objects" without identity. This means that a Grace implementation can support value objects using whatever implementation is most efficient: either passing by reference always, by passing some times by value, or even by inlining fields into their containing objects, and updating the field if the containing object assigns a new value.

11 Types

Grace uses structural typing [11, 32, 16]. Types primarily describe the requests objects can answer. Fields do not directly influence types, except in so far as a field with publicly-visible accessor methods will cause those methods to be part of the type (and in general to be visible to unconstrained clients).

Unlike in other parts of Grace, Type declarations are always statically typed, and their semantics may depend on the static types. The main case for this is determining between identifiers that refer to types, and those that refer to constant name definitions (introduced by **def**) which are interpreted as Singleton types.

11.1 Basic Types

Grace's standard prelude defines the following basic types:

- Object — the common interface of most objects
- Boolean — methods for **true** and **false**
- Number — numbers
- String — strings, and individual characters
- Pattern — pattern used in match/case statements
- Dynamic — dynamically typed expressions. If no types are provided on method formal parameters, the types are taken as dynamic by default.

There is also a top type, which can be written {} as an empty object type.

11.2 Object Types

Object types give the type of objects' methods. The various Cat object and class descriptions (see §7) would produce objects that conform to an object type such as the following.

```
{
        colour -> Colour
        name -> String
        miceEaten -> Number
        miceEaten:= (_ : Number) -> None
}
```

For commonality with method declarations, method arguments may be given both names and types within type declarations. A single identifier is interpreted as a formal parameter name with type Dynamic.

11.3 Type Declarations

Types — and generic types — may be named in type declarations:

```
type MyCatType ={ color -> Colour; name -> String }
   // I care only about names and colours

type MyGenericType<A,B> =
   where A <: Hashable; where B <: disposable
  {
   hashStore(_:A,_:B) -> Boolean // pity not just (A,B)
   cleanup(_:B)
  }
```

Grace has a single namespace: types live in the same namespace as methods and variables.

11.4 Relationships between Types — Conformance Rules

The key relation between types is **conformance**. We write B <: A to mean B conforms to A; that is, that B is a subtype of A, A is a supertype of B. This section draws heavily on the wording of the Modula-3 report [11], with apologies to Luca Cardelli et al.

If B <: A, then every object of type B is also an object of type A. The converse does not apply.

If A and B are ground object types, then B <: A iff

- B contains every method in A
- Every B method must have the same number of arguments as A, with the same distribution in multi-part method names.

- Every method with parameters "$(P_1, ...P_n) -> R$" in A must have a corresponding method in B "$(Q_1, ...Q_n) -> S$".
  - Argument types may be contravariant: $P_i <: Q_i$
  - Results types may be covariant: S <: R

If a class or object B inherits from another class A, then B's type should conform to A's type. If A and B are generic classes, then similar instantions of their types should conform.

The conformance relationship is used in **where** clauses to constrain formal generic type parameters of classes and methods.

11.5 Any and None

The type Any is the supertype of all types — and may also be written as {}.

The type None is the subtype of all types. There are no instances of None. In particular, neither undefined and nor any kind of nil are instances of None.

What happens if a method requested via Dynamic returns None, but the caller attempts to use that None value"

11.6 Variant Types

Untagged, retained variant types, written T1 | T2 ... | Tn , may refer to an object of any one of their component types. Instances of variant are not themselves tagged as members of the variant, however instances can be distinguised via their reified type information.

The only methods that may be requested via a variant type are methods with exactly the same declaration across all members of the variant. (Option) methods with different signatures may be requested at the most most specific argument types and least specific return type.

Variant types are retained as variants: they are *not* equivalent to the object type which describes all common methods. This is so that the exhaustiveness of match/case statements can be determined statically. In detail:

S <: S | T; T <: S | T
(S' <: S) & (T' <: T) ⟹ (S' | T') <: (S | T)

11.7 Intersection Types

(option) Intersection types, written T1 & T2 & ... &Tn, may refer to an object that conforms to all of the component types. The main use of intersection types is as bounds on where clauses.

```
class Happy<T>
   where T <: (Comparable<T> & Printable & Happyable)
{ foo ->

}
```

11.8 Union Types

(option) Structural union types (sums), written T1 + T2 + ... + Tn, may refer to an object that conforms to any of the component types. Unions are mostly included for completeness: variant types subsume most uses.

11.9 Type subtraction

(option) A type written T1 − T2 has the interface of T1 without any of the methods in T2.

11.10 Singleton Types

The names of singleton objects, typically declared in object declarations, may be used as types. Singleton types match only their singleton object. Singleton typs can be distinguised from regular types because Grace type declarations are statically typed.

```
def null = object { method isNull -> Boolean {return true} }

class Some<T> { thing : T ->
                    method isNull -> Boolean {return false} }

type Option<T> =Some<T> | null
```

11.11 Nested Types

(Option) Types may be nested inside types, written T1.T2

In this way a type may be used as a specification module.

11.12 Additional Types of Types

(option) Grace may support nullable types (written ?Type, defined as (Type|null)) and exact types (written =Type)

(option) Grace probably will support Tuple types, probably written Tuple<T1, T2... Tn>. We're not yet sure how.

11.13 Syntax for Types

This is very basic - but hopefully better than nothing!

```
Type := GroundType | (GroundType ("|" | "&") GroundType)...
GroundType ::= BasicType | BasicType "<" Type "," ... ">" | "Selftype"
BasicType ::= TypeID | "=" TypeID | "?" TypeID | "?=" TypeID
```

11.14 Reified Type Information Metaobjects and Type Literals

(option) Types are represented by objects of type Type (Hmm, should be Type<T>?). Since Grace has a single namespace, so types can be accessed by requesting their names.

To support anonymous type literals, types may be written in expressions: **type** Type. This expression returns the type metaobject repesenting the literal type.

11.15 Type Assertions

(option) Type assertions can be used to check conformance and equality of types.

```
assert {B <: A}
    // B 'conforms to' A.
    // B is a subtype of A
assert {B <: {foo(_:C) -> D}
    // B had better have a foo method from C returning D
```

11.16 Notes

1. (**Option**) Classes define a type (of the same name) — currently **this is NOT part of Grace**

2. (**Sanity Check**) these rules

3. (**To be done**) add in path types, types in objects.
4. What's the relationship between "type members" across inheritance (and subtyping???)
5. Classes are not types — are we sure about this?
6. Types are patterns (need to be to be matched against!)
7. Reified Generics formals are also patterns (see above)
8. On matching, How does destructuring match works? What's the protocol? Who defines the extractor method? (not sure why this is here)
9. Somehow, do classes need to define a type that describes the objects that are created by their factory methods.
10. Note that Generic Types use angle brackets, viz. ImmutableCollection<Figure>
11. can a type extend another type?
12. where do where clauses go?
13. method return types
14. Structural typing means we neither need nor want any variance annotations! Becaue Grace is structural, programmers can always write an (anonymous) structural type that gives just the interface they need — or such types could be stored in a library.
15. Should ObjectTypes permit formal parameter names or not? §11.2?
16. What actually gets returned from None? §6.3 §11.5
17. Tuples §11.12. Syntax as a type? Literal Tuple Syntax?
18. Nesting.

12 Pragmatics

The distribution medium for Grace programs, objects, and libraries is Grace source code.

Grace source files should have the file extension .grace. If, for any bizzare reason a trigraph extension is required, it should be .grc

Grace files may start with one or more lines beginning with #: these lines are ignored.

13 Libraries

13.1 Collections

Grace will support some collection classes.

Collections will be indexed `1.. size` by default; bounds should be able to be chosen when explicitly instantiating collection classes.

Acknowledgements

The Scala language specification 2.8 [37] and the Newspeak language specification 0.05 [6] were used as references for early versions of this document. The design of Grace (so far!) has been influenced by Algol [39, 36], AspectJ [27], BCPL [40], Beta [31], Blue [28, 29, 30], C [26], C++ [41], C$\sharp$ [4, 3], Eiffel [33, 34], Emerald [5], F_1 [10], $F\sharp$ [43], FGJ [22], $FJ\vee$ [23], FORTRESS [1], gBeta [13], Haskell [21], Java [12, 17], Kevo [44], Lisp [15], ML [35], Modula-2 [48], Modula-3 [11], Modular Smalltalk [47], Newspeak [8, 6], Pascal [25], Perl [46], Racket [14], Scala [38, 37], Scheme [42], Self [45], Smalltalk [18, 24, 9, 7], Object-Oriented Turing [20], Noney [32], Whiteoak [16]

at least: we apologise if we've missed any languages out. All the good ideas come from these languages: the bad ideas are all our fault [19].

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 numeric types
2. Block::apply §5 — How should we spell "apply"? "run"?
3. confirm method lookup algorithm, in particular relation between lexical scope and inheritance §8 ("Out then Up"). Is that enough? Does the no-shadowing rule work? If it does, is this a problem?
4. confirm "super" or other mechaism for requesting overridden methods §8.6
5. confirm rules on named method argument parenthesization §8.1
6. how are (mutually) recursive names initialised?
7. make the **def** keyword optional, or remove it, or return to **const** §6.1 post 10/02/2011.
8. support multiple constructors for classes §7.2
9. where should we draw the lines between object constructor expressions/named object declarations, class declarations, and "hand-built" classes? §7.3
10. what's the difference between **class** FOO {} and **def** FOO =**class** {} (for various values of "class")
11. how do factories etc relate to "uninitialized" §6.2
12. decide what to do about equality operators §10
13. Support for identifying static type decltype and dynamic type typeid/foo.getType
14. Support for type test (like instanceof) and static casts.
15. What is the namespace of types? What is the syntax of types? §11 More to the point, what is the type system?
16. Multiple Assignment §6.2 ? f<T> ?

17. Type assertions — should they just be normal assertions between types? so e.g. <: could be a normal operator between types...?

18. Grace needs subclass compatibility rules

19. BRANDS. Brand Brand Brand.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0β. Technical report, Sun Microsystems, Inc., March 2007.

[2] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), October 1993.

[3] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C♯. In *OOPSLA*, 2007.

[4] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C♯. In *ECOOP*, 2010.

[5] Andrew P. Black, Eric Jul, Norman Hutchinson, and Henry M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*. ACM Press, 2007.

[6] Gilad Bracha. Newspeak programming language draft specification version 0.0. Technical report, Ministry of Truth, 2009.

[7] Gilad Bracha and David Griswold. Stongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*. ACM Press, 1993.

[8] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda6. Modules as objects in Newspeak. In *ECOOP*, 2010.

[9] Tim Budd. *A Little Smalltalk*. Addison-Wesley, 1987.

[10] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *Computer Science Handbook*, chapter 97. CRC Press, 2nd edition, 2004.

[11] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 reference manual. Technical Report Research Report 53, DEC Systems Research Center (SRC), 1995.

[12] Carlton Egremont III. *Mr. Bunny's Big Cup o' Java*. Addison-Wesley, 1999.

[13] Erik Ernst. Family polymorphism. In *ECOOP*, 2001.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How To Design Programs.* MIT Press, 2001.

[15] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, 1991.

[16] Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA*, 2008.

[17] Brian Goetz, Time Peierls, Joshua Block, Joesph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison Wesley Professional, 2006.

[18] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[19] C.A.R. Hoare. Hints on programming language design. Technical Report AIM-224, Stanford Artificial Intelligence Laboratory, 1973.

[20] Ric Holt and Tom West. OBJECT ORIENTED TURING REFERENCE MANUAL seventh edition version 1.0. Technical report, Holt Software Associates Inc., 1999.

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *History of Programming Languages III*, pages 12–1–12–55. ACM Press, 2007.

[22] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[23] Atsushi Igarashi and Hideshi Nagira. Union types for object- oriented programming. *Journal of Object Technology*, 6(2):31–45, February 2007. http://www.jot.fm/issues/issues 2007 02/article3.

[24] Daniel H.H. Ingalls. Design principles behind Smalltalk. *BYTE Magazine*, August 1981.

[25] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer, 1975.

[26] Brian W. Kernighan and Dennis M. Ritchie. *The "C" Programming Language.* Addison-Wesley, 2nd edition, 1993.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[28] Michael Kölling, Bett Koch, and John Rosenberg. Requirements for a first year object-oriented teaching language. In *ACM Conference on Computer Science Education (SIGCSE)*, 1995.

[29] Michael Kölling and John Rosenberg. Blue — a language for teaching object-oriented programming. In *ACM Conference on Computer Science Education (SIGCSE)*, 1996.

[30] Michael Kölling and John Rosenberg. Blue — a language for teaching object-oriented programming language specification. Technical Report TR97-13, Monash University Department of Computer Science and Software Engineering, 1997.

[31] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[32] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP*, 2008.

[33] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[34] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[35] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[36] Peter Naur. The European side of the development of ALGOL. In *History of Programming Languages I*, pages 92–139. ACM Press, 1981.

[37] Martin Odersky. The Scala language specification version 2.8. Technical report, Programming Methods Laboratory, EFPL, July 2010.

[38] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[39] Alan J. Perlis. The American side of the development of ALGOL. In *History of Programming Languages I*, pages 75–91. ACM Press, 1981.

[40] Martin Richards and Colin Whitby-Stevens. *BCPL: the language and its compiler*. Cambridge University Press, 1980.

[41] Bjarne Stroustrup. Why C++ is not just an object-oriented programming language. In *OOPSLA Companion*. ACM Press, 1995.

[42] Gerald Sussman and Guy Steele. SCHEME: An interpreter for extended lambda calculus. Technical Report AI Memo 349, MIT Artificial Intelligence Laboratory, December 1975.

[43] Don Syme. The F♯ draft language specification. Technical report, Microsoft, 2009.

[44] Antero Taivalsaari. Delegation versus concatenation or cloning is inheritance too. *OOPS Messenger*, 6(3), 1995.

[45] David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

[46] Larry Wall. Perl, the first postmodern computer language. `http://www.wall.org/ larry/pm.html`, Spring 1999.

[47] Allen Wirfs-Brock and Brian Wilkerson. Modular Smalltalk. In *OOPSLA*, 1998.

[48] Niklaus Wirth. Modula-2 and Oberon. In *HOPL*, 2007.