The Grace Programming Language Draft Specification Version 0.3.53

Andrew P. Black
Portland State University, black@cs.pdx.edu

Kim B. Bruce

James Noble

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

Part of the Programming Languages and Compilers Commons

Citation Details
Black, Andrew P.; Bruce, Kim B.; and Noble, James, "The Grace Programming Language Draft Specification Version 0.3.53" (2013). Computer Science Faculty Publications and Presentations. 93.
https://pdxscholar.library.pdx.edu/compsci_fac/93

This Working Paper is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.
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:

- collection syntax and collection literals
- tuples vs multiple values vs multiple returns
- nested static type system (although we’ve made a start)
- encapsulation system (privacy annotations)
- module system
- metadata (Java’s @annotations, C♯ attributes, final, abstract etc)
- purity and non-nulls.
- reflection
- assertions, data-structure invariants, pre & post conditions, contracts
- regexp
- 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.


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.


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:

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

code without punctuation:

```java
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 syntactically).

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 and Operators

Grace has the following reserved words and reserved operators. The '?' indicates words related to design options not yet chosen.

    assert case catch class const def extends false finally method object outer(?) prefix raise return self Selftype 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 supports a single type Number. Number maintains 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 are written using && 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 andAlso { Q }
toBe orElse { ! toBe }
```
4.3 Strings and Characters

String literals in Grace are written between double quotes. 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 includes mechanisms to support efficient incremental string construction.

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>double quote</td>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>backslash</td>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\n</td>
<td>line-feed</td>
<td>\r</td>
<td>carriage-return</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
<td>\l</td>
<td>unicode newline</td>
</tr>
<tr>
<td>\f</td>
<td>page down</td>
<td>\e</td>
<td>escape</td>
</tr>
<tr>
<td>\f</td>
<td>left bracket</td>
<td>}</td>
<td>right bracket</td>
</tr>
<tr>
<td>(space)</td>
<td>non-breaking space</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Grace string escapes.

Examples

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

4.4 String interpolation

Within a string literal, Grace treats expressions enclosed in braces specially. The expression is evaluated, the asString method is requested on the resulting object, and the resulting string is inserted into the string literal in place of the brace expression.

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 \( \rightarrow \) symbol.

\[
\begin{align*}
\{ & \text{do.something} \} \\
& \{ \ i \rightarrow i+1 \} \\
& \{ \ \text{sum, next} \rightarrow \text{sum +next} \}
\end{align*}
\]

Blocks construct objects with a single method named \texttt{apply}, \texttt{apply(n)}, \texttt{apply(n, m)}, \ldots depending on the number of parameters. Requesting the \texttt{apply} method evaluates the block. It is an error to provide the wrong number of arguments.

\[
\text{for (1..10) do \{ \\
\quad i \rightarrow \text{print } i \\
\text{\}}}
\]

might be implemented as

\[
\text{\textbf{method} for (collection) do (block) \{ \\
\quad \ldots \\
\quad \text{block.apply( collection .at(i))} \\
\quad \ldots \\
\text{\}}}
\]

Here is another example:

\[
\begin{align*}
\textbf{var} & \text{ sum := 0} \\
\textbf{def} & \text{summingBlock : Block<Number,Number> =} \\
& \{ \ i: \text{Number} \rightarrow \text{sum := sum +i} \} \\
\text{summingBlock.apply(4)} & \quad // \text{sum is now 4} \\
\text{summingBlock.apply(32)} & \quad // \text{sum is now 36}
\end{align*}
\]

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.

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**

```plaintext
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 and types and ...). 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**

```plaintext
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 “\( x:=(1) \)” 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 are named “prefix” followed by the operator character(s).

Methods may have “repeated parameters” to allow them to take a variable number of parameters, that is to provide what is often called “variable arity” or “varargs”. A repeated parameters, if present, must be the last parameter in a part of a multipart method name. Repeated parameters are designated by a star “∗” before the name of the parameter. Inside the method, a repeated parameter has the type of an immutable collection of the declared type—for example, a parameter declared `∗args : String` has the type `args : ImmutableCollection<String>`.

Methods may be declared with generic type parameters, in which case they must be requested with generic type arguments. Generic type parameters may be constrained 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 object-level expressions, initializer expressions and method bodies are executed in that lexical context. Each time an object literal is executed, a new object is created.
A constant can be defined by an object literal, such as:

```python
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` 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 that creates “instances of the class”. A class declaration is syntactically similar to the method declaration for the factory method, where the body of the factory method is the body of an object literal:

#### Examples

```python
class aCat.ofColour ( aColour ) named (aName) {
    def colour : Colour = aColour
    def name : String = aName
    var miceEaten := 0
}
```

The `ofColour()named()` method takes two arguments; the object that it returns has the fields and methods listed in the body of the constructor that follows the name of the factory method. 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 arguments to `.ofColour()named()`:

```python
def fergus = aCat.ofColour "tortoiseshell" named "Fergus Trouble"
```

If the programmer wants a factory object with more than one method, she is free to build such an object using nested object constructors. The above declaration for `class aCat` is equivalent (modulo types and modules) to the following nested object declarations:

```python
def aCat = object { // the cat factory
    def colour : Colour = Colour.tabby
    def name : String = "Unnamed"
    var miceEaten := 0
}
```
method ofColour (aColour: Colour) named (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 aCat.ofColour()named(), not the factory object aCat.

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 aPedigreeCat.ofColour (aColour) named (aName) {
    inherits Cat.ofColour (aColour) named (aName)
    var prizes := 0
    <override> method miceEaten {0};
    <override> method miceEaten := (n:Number) {return} //ignore attempts to change it
}

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

```plaintext
def aPedigreeCat = object { // the cat factory
    method ofColour (aColour: Colour) named (aName: String) −> PedigreeCat {
        object { // the cat herself
            def colour : Colour := aColour
            def name : String := aName
            <<private>> var _miceEaten := 0 // ugly. super—ugly
            var _prizes := 0
            method miceEaten := (n: Number) { return } // ignore attempts to change it
        } // object
    } // method
} // object
```

7.5 Generic Classes

Classes may optionally be declared with generic type parameters. The corresponding requests on the factory methods must be provided with type arguments. Generic type parameters may be constrained with `where` clauses.

Examples

```plaintext
class aVector. ofSize ( size )<T> {  
    var contents := Array. size ( size )
    method at(index : Number) −> T { return contents.at() }
    method at(index : Number) put(elem : T) { }
}
```

```plaintext
class aSortedVector. ofSize<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 of method names.
Parenthesis may be omitted where they would enclose a single argument,
provided that argument is a literal other than an object literal.

8.2 Assignment Methods
A assignment method request is an explicit receiver followed by a dot, then
an identifier, followed by “:=”, and then a single argument.

Examples

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

Assignment methods 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 parenthe-
sis 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 // evaluates to 7
1 += 4 − *= 4 // syntax error

Named method requests without arguments bind more tightly than op-
erator method requests. The following examples show first the Grace ex-
pressions as they would be written, followed by the parse.

Examples

1 + 2. i
(a * a) + (b * b). sqrt
((a * a) + (b * b)). sqrt
a * a + b * b
a + b + c
a − b − c

1 + (2. i)
(a * a) + ((b * b). sqrt)
((a * a) + (b * b)). sqrt
(a * a) + (b * b)
(a + b) + c
(a − b) − c

8.4 Unary Prefix Operator Method

Grace supports unary prefix operator methods: since Grace does not sup-
port 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

- \((b + (4 \times a).\sqrt{\cdot})\)
- \(b.squared\)    // illegal
- \((-b).squared\)
- \((-b.squared)\)

\[\text{status .ok := !(engine.isOnFire) && wings.areAttached && isOnCourse}\]

8.5 Accessing Operator Method

Grace supports an operator \([]\) which can be defined in libraries, *e.g.*, for indexing collections.

Examples

\[\text{print ( a[3] )} \quad \text{// requests method \([]\) on a with argument 3}\]

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

Examples

\[a[3] := \text{"Hello"} \quad \text{// requests method \([]:=?\) on a with arguments 3 and \text{"Hello"}}\]

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

8.6 Super Requests

The reserved word \(\text{super}\) may be used only as an explicit receiver. In over-riding methods, method requests with the pseudo-receiver \(\text{super}\) request the prior overridden method with the given name from \(\text{self}\). Note that no “search” is involved; super-requests can be resolved statically, unlike other method requests.

Examples

\text{\texttt{super.foo}}
\text{\texttt{super.bar(1,2,6)}}
\text{\texttt{super.doThis(3) timesTo("foo")}}
\text{\texttt{super + 1}}
\text{\texttt{!super}}

\[\text{foo(super)} \quad \text{// syntax error}\]
\[1 + \text{super} \quad \text{// syntax error}\]
8.7 Encapsulation

The design of Grace’s encapsulation system has not yet begun in earnest. Grace will use metadata annotations to allow the declaration of confidential methods, that is, 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 are inferred.

Examples

\[
\text{sumSq}\langle\text{Integer64}\rangle(10.\text{i64}, 20.\text{i64})
\]
\[
\text{sumSq}(10.\text{i64}, 20.\text{i64})
\]

9 Control Flow

Control flow statements in Grace are syntactically method requests. 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:

\[
\text{if (test) then } \{\text{block}\}
\]
\[
\text{if (test) then } \{\text{block}\} \text{ else } \{\text{block}\}
\]

While statement:

\[
\text{while } \{\text{test}\} \text{ do } \{\text{block}\}
\]

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

Grace supports a match/case construct. Match takes one argument and matches it against a series of blocks introduced by “case”. Pattern matching supports 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 anything
  case { _ -> print("did not match") }

9.2.1 API Design — Scary Overkill Monadic Version (under discussion)

Pattern matching is based around the Pattern type:

def type Pattern<R,X> =

match(o : Any) -> MatchResult<R,X>
} where X <: Tuple

type MatchResult<R,X> = {
  succeeded -> Boolean
  next -> N
  result -> R
  bindings -> X
} where X <: Tuple

A pattern can test if any object matches the pattern, returning a `MatchResult` which is either a `SuccessfulMatch` or a `FailedMatch`. From a successful match, the `result` is the return value, typically the object matched, and the `bindings` are a tuple of objects that may be bound to intermediate variables, generally used for destructuring objects. If a prefix of the object is matched, any unmatched objects are returned in `next`.

A type declaration creates a singleton object that acts as a pattern. If the type has an `extract` method that returns a tuple, `X` is the return type of that method; if not, `X` is `None`.

For example, this `Point` type:

type Point = {
  x -> Number
  y -> Number
  extract -> Tuple<Number,Number>
}

implemented by this `CartesianPoint` class:

class CartesianPoint .new(x' : Number, y' : Number) -> Point {
  def x = x'
  def y = y'
  def extract = [x,y]
}

then these hold:

def cp := CartesianPoint .new(10,20)

Point.match(cp).bindings // returns [10, 20]
Point.match(true) // returns MatchFailure
9.2.2 Translating Matching-blocks

Matching-blocks are blocks with one formal parameter. This parameter may be a pattern, rather than just being a fresh variable (potentially with a type). Matching-blocks are themselves patterns: one-argument (matching) block with parameter type $A$ and return type $R$ also implements $\text{Pattern}<R,\text{None}>$.

A recursive, syntax-directed translation maps matching-blocks into blocks with separate explicit patterns non-matching blocks that are called via apply only when their patterns match.

First, the matching block is flattened — translated into a straightforward non-matching block with one parameter for every bound name or placeholder. For example:

$$
\{ \_ : \text{Pair}(a, \text{Pair}(b, c)) \rightarrow \{a\} \{b\} \{c\} \}
$$

is flattened into

$$
\{ \_, a, b, c \rightarrow \{a\} \{b\} \{c\} \}
$$

then the pattern itself is translated into a composite object structure:

```python
def mypat =
  MatchAndDestructuringPattern.new(Pair,
    VariablePattern.new("a"),
    MatchAndDestructuringPattern.new(Pair,
      VariablePattern.new("b"), VariablePattern.new("c")))
```

Finally, the translated pattern and block are glued together via a $\text{LambdaPattern}$:

```python
LambdaPattern.new( mypat, \{ \_, a, b, c \rightarrow \{a\} \{b\} \{c\} \})
```

The translation is as follows:

<table>
<thead>
<tr>
<th>$e$</th>
<th>$[[e]]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$_ : e$</td>
<td>$[[e]]$</td>
</tr>
<tr>
<td>$_ - v$ (fresh, unbound variable)</td>
<td>$\text{WildcardPattern}$</td>
</tr>
<tr>
<td>$_ v$ (bound variable)</td>
<td>$\text{VariablePattern}(&quot;v&quot;)$</td>
</tr>
<tr>
<td>$_ v : e$</td>
<td>$\text{error}$</td>
</tr>
<tr>
<td>$e(f, g)$</td>
<td>$\text{AndPattern}.new(\text{VariablePattern}.new(&quot;v&quot;),[[e]])$</td>
</tr>
<tr>
<td>literal</td>
<td>$\text{MatchAndDestructuringPattern}.new(e, [[f]], [[g]])$</td>
</tr>
<tr>
<td>$e$ not otherwise translated</td>
<td>$\text{literal}$</td>
</tr>
</tbody>
</table>
9.2.3 Implementing Match-case

Finally the match(1)*case(N) methods can be implemented directly, e.g.:

```java
method match(o : Any)
    case(b1 : Block<B1,R>)
    case(b2 : Block<B2,R>)
    {
        for [b1, b2] do { b ->
            def rv = b.match(o)
            if (rv.succeeded) then {return rv.result}
        }
    }
    FailedMatchException.raise
}
```

or (because matching-blocks are patterns) in terms of pattern combinators:

```java
method match(o : Any)
    case(b1 : Block<B1,R>)
    case(b2 : Block<B2,R>)
    {
        def rv = (b1 || b2).match(o)
        if (rv.succeeded) then {return rv.result}
    }
    FailedMatchException.raise
}
```

First Class Patterns  While all types are patterns, not all patterns are types. For example, it would seems sensible for regular expressions to be patterns, potentially created via one (or more) shorthand syntaxes (shorthands all defined in standard Grace)

```java
match (myString)
    case { "" -> print "null string" }
    case { Regexp.new("[a-z]*") -> print "lower case" }
    case { "[A-Z]*.r -> print "UPPER CASE" }
    case { /[0-9]*" -> print "numeric" }
    case { ("Forename:([A-Za-z]*)Surname:([A-Za-z]*)".r2)(fn,sn) ->
            print "Passenger {fn. first } {sn}" }
```

With potentially justifiable special cases, more literals, e.g. things like tuples/lists could be destructured `[a,b,...] -> a * b. Although it would
be very nice, it’s hard to see how e.g. points created with "3@4" could be destructed like a@b -> print "x: \{a\}, y: \{b\}" without yet more bloated special-case syntax.

**Discussion** This rules try to avoid literal conversions and ambiguous syntax. The potential ambiguity is whether to treat something as a variable declaration, and when as a first-class pattern. These rules (should!) treat only fresh variables as intended binding instances, so a “pattern” that syntactically matches a simple variable declaration (as in this block { empty -> print "the singleton empty collection" }) will raise an error — even though this is unambiguous given Grace’s no shadowing rule.

Match statements that do nothing but match on types must distinguish syntactically from a variable declaration, e.g.:

```plaintext
match (rv)
  case { (FailedMatch) -> print "failed" }
  case { _ : SuccessfulMatch -> print "succeeded" }
```

while writing just:

```plaintext
match (rv)
  case { FailedMatch -> print "failed" }
  case { SuccessfulMatch -> print "succeeded" }
```

although closer to the type declaration, less gratuitous, and perhaps less error-prone, would result in two errors about variable shadowing.

**Self-Matching** For this to work, the main value types in Grace, the main literals — Strings, Numbers — must be patterns that match themselves. That’s what lets things like this work:

```plaintext
method fib(n : Number) {
  match (n)
    case { 0 -> 0 }
    case { 1 -> 1 }
    case { _ -> fib(n-1)+fib(n-2) }
}
```

With this design, there is a potential ambiguity regarding Booleans: “\texttt{true} || \texttt{false}” as an expression is very different from “\texttt{true | false}” as a composite pattern! Unfortunately, if Booleans are Patterns, then there’s no way the type checker can distinguish these two cases.
If you want to match against objects that are not patterns, you can
lift any object to a pattern that matches just that object by writing e.g. 
LiteralPattern .new(o) (option — or something shorter, like a prefix =˜?).
(Option) matches could also be written as an operator e.g. p =˜ q for 
p.match(q).succeeded. This might be good for scripts, but really it’s probably a very bad idea in general: that way lies Thorn.

9.3 Exceptions (under discussion)
Grace supports basic unchecked exceptions. Exceptions are generated by 
requesting the raise method from an Exception class:

UserException.raise("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 termi-
nated 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 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”, have the same methods declared in the same lexical environments, 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) 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, 34, 17]. 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 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.

\{
  \text{colour} \rightarrow \text{Colour} \\
  \text{name} \rightarrow \text{String} \\
  \text{miceEaten} \rightarrow \text{Number} \\
  \text{miceEaten:} = (\_ : \text{Number}) \rightarrow \text{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:

\textbf{type} \text{MyCatType} =\{ \text{color} \rightarrow \text{Colour}, \text{name} \rightarrow \text{String} \} \\
// I care only about names and colours

\textbf{type} \text{MyGenericType}\langle A,B \rangle = \\
\textbf{where} A <: \text{Hashable}, \textbf{where} B <: \text{disposable} \\
\{ \\
  \text{hashStore}(\_:A,\_:B) \rightarrow \text{Boolean} // pity not just (A,B) \\
  \text{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 \textbf{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 $"(P_1, \ldots, P_n) \rightarrow R"$ in $A$ must have a corresponding method in $B$ $"(Q_1, \ldots, Q_n) \rightarrow S"$.
  - 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 $\textit{where}$ clauses to constrain formal generic type parameters of classes and methods.

11.5 $\textbf{Any and None}$

The type $\textbf{Any}$ is the supertype of all types — and may also be written as $\{\}$. The type $\textbf{None}$ is the subtype of all types. There are no instances of $\textbf{None}$. In particular, neither undefined nor any kind of nil is an instance of $\textbf{None}$.

$\textbf{What happens if a method requested via Dynamic returns None, but the caller attempts to use that None value?}$

11.6 $\textbf{Variant Types}$

Variables with untagged, retained variant types, written $T_1 \mid T_2 \ldots \mid T_n$, may refer to an object of any one of their component types. No objects actually have variant types, only variables. The actual type of an object referred to by a variant variable can be determined using that object’s 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:

\[
S <: S \mid T; \quad T <: S \mid T \\
(S' <: S) \land (T' <: T) \Rightarrow (S' \mid T') <: (S \mid T)
\]

### 11.7 Intersection Types

(option) An object conforms to an Intersection type, written \(T_1 \& T_2 \& \ldots \& T_n\), if and only if that object conforms to all of the component types. The main use of intersection types is as bounds on where clauses.

```plaintext
class Happy<T>
    where T <: (Comparable<T> & Printable & Happyable)
    { p -> ...
    }
```

### 11.8 Union Types

(option) Structural union types (sums), written \(T_1 + T_2 + \ldots + T_n\), 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 \(T_1 - T_2\) has the interface of \(T_1\) without any of the methods in \(T_2\).

### 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 types can be distinguished from regular types because Grace type declarations are statically typed.

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

class Some<T> { thing : T ->
    method isNull -> Boolean {return false} }
```
\textbf{type} Option<T> = Some<T> | null

11.11 Nested Types

(Option) Types may be nested inside types, written \texttt{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 \texttt{?Type}, defined as \texttt{(Type | null)})

and exact types (written \texttt{= Type})

(Option) Grace probably will support Tuple types, probably written \texttt{Tuple<T1, T2... Tn>}. We're not yet sure how.

(Option) Grace may support selftypes, written \texttt{Selftype}.

11.13 Syntax for Types

This is very basic - but hopefully better than nothing!

\begin{align*}
\text{Type} & := \text{GroundType} \mid (\text{GroundType } ("|" \mid "&") \text{ GroundType})...
\text{GroundType} & := \text{BasicType} \mid \text{BasicType } "<" \text{ Type }, ">" \mid "\text{ Selftype}"
\text{BasicType} & := \text{TypeID} \mid "=" \text{TypeID} \mid "?" \text{TypeID} \mid "?=" \text{TypeID}
\end{align*}

11.14 Reified Type Information Metaobjects and Type Literals

(Option) Types are represented by objects of type \texttt{Type} (Hmm, should be \texttt{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: \texttt{type Type}. This expression returns the type metaobject representing the literal type.

11.15 Type Assertions

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

\begin{verbatim}
assert {B <: A}
  // B 'conforms to' A.
  // B is a subtype of A
assert {B <: {foo(_:C) -> D}}
\end{verbatim}
// 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! Because Grace is structural, programmers can always write an (anonymous) structural type that gives just the interface they need — or such types could be storted 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
19. Serialization
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 bizarre 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.

12.1 Garbage Collection

Grace implementations should be garbage collected. Safepoints where GC may occur are at any backwards branch and at any method request.

Grace will not support finalisation.

12.2 Concurrency and Memory Model

The core Grace specification does not describe a concurrent language. Different concurrency models may be provided as dialects.

Grace does not provide overall sequential consistency. Rather, Grace provides sequential consistency within a single thread. Across threads, any value that is read has been written by some thread sometime — but Grace does not provide any stronger guarantee for concurrent operations that interfere.

Grace’s memory model should support efficient execution on architectures with Total Store Ordering (TSO).

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

Thanks to Josh Bloch, Cay Horstmann, Micahel Kølling, Doug Lea, the participants at the Grace Design Workshops and the IFIP WG2.16 Programming Language Design for discussions about the language design.
Thanks to Michael Homer and Ewan Tempero for their comments on drafts.

The Scala language specification 2.8 [39] 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 [41, 38], AmbientTalk [12], AspectJ [29], BCPL [42], Beta [33], Blue [30, 31, 32], C [28], C++ [43], C♯ [4, 3], Eiffel [35, 36], Emerald [5], F₁ [10], F♯ [45], FGJ [24], FJ ∨ [25], FORTRESS [1], gBeta [14], Haskell [22], Java [13, 18], Kevo [46], Lua [23], Lisp [16], ML [37], Modula-2 [50], Modula-3 [11], Modular Smalltalk [49], Newspeak [8, 6], Pascal [27], Perl [48], Racket [15], Scala [40, 39], Scheme [44], Self [47], Smalltalk [19, 26, 9, 7], Object-Oriented Turing [21], Noney [34], and Whiteoak [17] 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 [20].
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 mechanism 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. More to the point, what is the type system?
15. Multiple Assignment §6.2 ? f<T> ?
16. Type assertions — should they just be normal assertions between types? so e.g. <: could be a normal operator between types...?
17. Grace needs subclass compatibility rules
18. BRANDS. Brand Brand Brand.
19. weak references

B Grammar

```plaintext
// top level
def program = rule { codeSequence " rep(ws) " end}
def codeSequence = rule { repdel(( declaration | statement), semicolon) }
def innerCodeSequence = rule { repdel(( innerDeclaration | statement), semicolon) }

// declarations
def declaration = rule { varDeclaration | defDeclaration | classDeclaration | typeDeclaration | methodDeclaration }
def innerDeclaration = rule { varDeclaration | defDeclaration | classDeclaration | typeDeclaration }
def varDeclaration = rule { varId " identifier " opt(colon " typeExpression ) " opt( assign " expression ) }
def defDeclaration = rule { defId " identifier " opt(colon " typeExpression ) " equals " expression }
def methodDeclaration = rule { methodId " methodHeader " methodReturnType " whereClause " lBrace " innerCodeSequence " rBrace }
def classDeclaration = rule { classId " identifier " dot " classHeader " methodReturnType " whereClause " lBrace " inheritsClause " codeSequence " rBrace }

// warning: order here is significant!
def methodHeader = rule { accessingAssignmentMethodHeader | accessingMethodHeader | assignmentMethodHeader | methodWithArgsHeader | unaryMethodHeader | operatorMethodHeader | prefixMethodHeader }
def classHeader = rule { methodWithArgsHeader | unaryMethodHeader }
def inheritsClause = rule { opt( inheritsId " expression " semicolon ) }
def unaryMethodHeader = rule { identifier " genericFormals }
def methodWithArgsHeader = rule { firstArgumentHeader " repsep( argumentHeader(opt(ws)) ) }
def argumentHeader = rule { identifier " methodFormals }
def operatorMethodHeader = rule { otherOp " oneMethodFormal }
def prefixMethodHeader = rule { opt(ws) " token(" prefix") " otherOp } // forbid space after prefix?
def assignmentMethodHeader = rule { identifier " assign " oneMethodFormal }
def accessingAssignmentMethodHeader = rule { lBrack " assign " genericFormals " methodFormals }
```

def methodReturnType = rule { opt(arrow " nonEmptyTypeExpression ) }
def methodFormals = rule { lParen rep1sep( identifier " opt( colon \ typeExpression ), comma ) rParen }
def oneMethodFormal = rule { lParen identifier " opt( colon \ typeExpression ) rParen }
def blockFormals = rule { lParen rep1sep( identifier " opt( colon \ typeExpression ), comma ) rParen }
def matchBinding = rule { ( identifier | literal | parenExpression ) opt( colon \ nonEmptyTypeExpression opt( matchingBlockTail ) ) }
def matchingBlockTail = rule { lParen rep1sep( matchBinding, comma ) rParen }
def typeDeclaration = rule { typeId " identifier " genericFormals " equals " nonEmptyTypeExpression " semicolon " whereClause " }
def typeExpression = rule { ( opt( ws ) " typeOpExpression " opt( ws ) ) | opt( ws ) " typeOpExpression " opt( ws ) } 
def typeOp = rule { opSymbol("|") | opSymbol("&") | opSymbol("+") } 
// def typeOpExpression = rule { rep1sep( basicTypeExpression , typeOp ) } 
def typeOpExpression = rule { // this complex rule ensures two different typeOps have no precedence var otherOperator basicTypeExpression " opt( ws ) " opt( typeOp , { s -> otherOperator := s; true } ) " rep1sep( basicTypeExpression " opt( ws ) , guard( typeOp , { s -> s == otherOperator } ) ) } 
def basicTypeExpression = rule { nakedTypeLiteral | literal | pathTypeExpression | parenTypeExpression } 
// if we keep this, note that in a typeExpression context \{ a ; \} is interpreted as type \{ a ; \} 
// otherwise as the block \{ a ; \} 
def pathTypeExpression = rule { opt( superId " dot ) " rep1sep( \{ identifier " genericActuals \} , dot ) } 
def parenTypeExpression = rule { lParen \ typeExpression " rParen } 
// statements 
def statement = rule { returnStatement | ( expression " opt( assignmentTail ) ) } 
// do we need constraints here on which expressions can have an assignmentTail 
// could try to rewrite as options including \{ expression " arrayAccess " assignmentTail \} 
// expression " dot " identifier " assignmentTail 
def returnStatement = rule { symbol("return") " opt( ws ) " opt( expression ) } 
//doesn’t need parens 
def assignmentTail = rule { assign " expression } 
// expressions 
def expression = rule { opExpression } 
//def opExpression = rule { rep1sep( addExpression , otherOp ) }
def opExpression = rule { // this complex rule ensures two different otherOps have no precedence
  var otherOperator
  addExpression " opt(ws) "
  opt( guard(otherOp, { s -> otherOperator:= s; true }) ) "
  rep1sep(addExpression " opt(ws),
    guard(otherOp, { s -> s == otherOperator }) )
}

def addExpression = rule { rep1sep(multExpression, addOp) }
def multExpression = rule { rep1sep( prefixExpression, multOp) }
def prefixExpression = rule {
  (rep(otherOp) ˜ selectorExpression )
  | (rep1(otherOp) ˜ superId)
}

// we can have !super

def selectorExpression = rule {
  primaryExpression ˜ rep( selector )
}
def selector = rule {
  (dot " unaryRequest )
  | (dot " requestWithArgs )
  | (lBrack " rep1sep( expression ,comma ) " rBrack)
}

def operatorChar = CharacterSetParser .new("!@#$%ˆ&|~\+=−∗/><:").//had to be moved up

//special symbol for operators: cannot be followed by another operatorChar

method opsymbol(s : String) {
  trim(token(s) ˜ not(operatorChar))
}
def multOp = opsymbol "∗" | opsymbol "/

def addOp = opsymbol "＋" | opsymbol "−"

def otherOp = rule {
  guard(trim(rep1(operatorChar)), { s -> !parse(s) with( reservedOp " end ) ) })
  // encompasses multOp and addOp

def operator = rule { otherOp | reservedOp }

def unaryRequest = rule {
  trim( identifier ) " genericActuals " not(delimitedArgument) }
def requestWithArgs = rule {
  firstRequestArgumentClause " repsep(requestArgumentClause,opt(ws))
}
def firstRequestArgumentClause = rule {
  identifier " genericActuals " opt(ws) " delimitedArgument
}
def requestArgumentClause = rule {
  identifier " opt(ws) " delimitedArgument
}
def delimitedArgument = rule {
  argumentsInParens | blockLiteral | stringLiteral
}
def argumentsInParens = rule {
  lParen " rep1sep(drop(opt(ws)) " expression ,comma ) " rParen
}

def implicitSelfRequest = rule {
  requestWithArgs | rep1sep(unaryRequest.dot)
}
def primaryExpression = rule {
  literal | nonNakedSuper | implicitSelfRequest | parenExpression
}
def parenExpression = rule {
  lParen " rep1sep(drop(opt(ws)) " expression , semicolon ) " rParen
  // TODO should parenExpression be around a codeSequence?
}
def nonNakedSuper = rule {
  superId " not(not( operator | lBrack ))
}

// "generics"
def genericActuals = rule {
  opt(IGeneric " opt(ws) "
    rep1sep(opt(ws) " typeExpression " opt(ws),opt(ws) " comma " opt(ws)) "
    opt(ws) " rGeneric )
}
def genericFormals = rule {
  opt(IGeneric " rep1sep( identifier ,comma ) " rGeneric )
}
def whereClause = rule { repdel(whereId \_ typePredicate , semicolon ) }
def typePredicate = rule { expression }

// wherever genericFormals appear, there should be a whereClause nearby.

// " literals ">
def literal = rule { stringLiteral | selfLiteral | blockLiteral | numberLiteral | objectLiteral | tupleLiteral | typeLiteral }
def stringLiteral = rule { opt(ws) \" doubleQuote \" rep( stringChar ) \" doubleQuote \" opt(ws) }
def stringChar = rule { ( drop(backslash) \~ escapeChar ) | anyChar | space }
def blockLiteral = rule { lBrace \~ opt( ( matchBinding | blockFormals ) \~ arrow ) \~ innerCodeSequence \~ rBrace }
def selfLiteral = symbol " self"
def numberLiteral = trim( DigitStringParser .new)ndef objectLiteral = rule { objectId \~ lBrace \~ inheritsClause \~ codeSequence \~ rBrace }
def tupleLiteral = rule { lBrack \~ repsep( expression , comma ) \~ rBrack }
def typeLiteral = rule { typeId \~ opt(ws) \~ nakedTypeLiteral }
def nakedTypeLiteral = rule { lBrace \~ opt(ws) \~ repdel( methodHeader \~ methodReturnType , (semicolon | whereClause) ) \~ opt(ws) \~ rBrace }

// terminals
def backslash = token "\\" // doesn’t belong here, doesn’t work if left below!
def doubleQuote = token "\"
def space = token " ">
def colon = rule { both(symbol " ", not(assign)) }
def newLine = symbol "\n"
def lParen = symbol "(">
def rParen = symbol ")"
def lBrace = symbol "{"
def rBrace = symbol "}"
def IBrack = symbol "["]
def rBrack = symbol "]"]"
def lrBrack = symbol "[]"
def arrow = symbol "\−\>",ndef dot = symbol ".",ndef assign = symbol ":=",
def equals = symbol "="
def IGeneric = token "<",
def rGeneric = token ">"
def comma = rule { symbol(\,,\,) }
def escapeChar = CharacterSetParser .new(azChars ++ AZChars ++ otherChars )ndef azChars = "abcdefghijklmnopqrstuvwxyz"ndef AZChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"ndef otherChars = "1234567890\@#\%\&\*\(\)\−\=\[\]\|\:\;\<\>,./"
def anyChar = CharacterSetParser .new(azChars ++ AZChars ++ otherChars)
def identifierString = trim(GraceIdentifierParser .new)

// def identifier = rule { bothAll(trim( identifierString ), not( reservedIdentifier )) }  
// bothAll ensures parses take the same length
// def identifier = rule{ both( identifierString , not( reservedIdentifier )) }  
// both doesn’t ensure parses take the same length
def identifier = rule{ guard( identifierString , { s ! parse(s) with( reservedIdentifier ~ end ) })}
// probably works but runs out of stack

// anything in this list needs to be in reservedIdentifier below (or it won’t do what you want)
def superId = symbol "super"
def extendsId = symbol "extends"
def inheritsId = symbol "inherits"
def classId = symbol "class"
def objectId = symbol "object"
def typeId = symbol "type"
def whereId = symbol "where"
def defId = symbol "def"
def varId = symbol "var"
def methodId = symbol "method"
def prefixId = symbol "prefix"
def interfaceld = symbol "interface"

def reservedIdentifier = rule { selfLiteral | superId | extendsId | inheritsId | 
  classId | objectId | typeId | whereId | 
  defId | varId | methodId | prefixId | interfaceld }  // more to come

def reservedOp = rule { assign | equals | dot | arrow | colon | semicolon }  // this is not quite right
References


[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.


