The Grace Programming Language
Draft Specification Version 0.1

Andrew P. Black   Kim B. Bruce   James Noble

June 4, 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
- static type system
- encapsulation system
- module system
- metadata (Java's @annotations, C$\sharp$ attributes, final, abstract etc)
- purity and non-nulls.
- reflection
- assertions, data-structure invariants, pre & post conditions, contracts
- regexps
- test support
- libraries

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

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

2 User Model

All designers in fact have user and use models consciously or subconsciously in mind as they work. Team design. . . requires explicit models and assumptions.

Frederick P. Brooks, *The Design of Design.* 2010.

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

3 Syntax

Much of the following text assumes the reader has a minimal grasp of computer terminology and a "feeling" for the structure of a program.

Kathleen Jensen and Niklaus Wirth, *Pascal: User Manual and Report.*

Grace programs are written in 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 preceeding syntactic unit; comments following a blank line are attached to the largest immediately following syntatic 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.

Any sequence of characters surrounded by back-quotes "‘" is an identifer (following F♯)

Examples

‘Java$$name(34)‘ ‘ illegal identifier ‘

3.4 Reserved Words

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

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

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

All numeric types are sub-types of an abstract type Number. All Grace implementations must support both exact Rational and Float64, the latter being Grace's name for the IEEE 754 standard's "Binary64" floating point numbers. 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.

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.not }
```

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.
\ (\ 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() 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
const 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

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

6.1 Constants

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

Examples

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

Grace has a single namespace for methods and constants (and variables). A constant declaration of x can be seen as creating a (nullary) 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.
var x:Binary64 := 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. The accessors for an instance variable is hidden by the declaration of a temporary with the same name; they can still be accessed using an *explicit* method request on **self**. That is, **self**.x is always a request for **self** to execute the method x, even when x is a temporary variable.

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

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 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** Unit, 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 ":=".

(Option) If supported, prefix operator methods will be named "**prefix**" followed by the operator character(s).

Examples

```
method pi {3.141592634}

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

method +(other : Point) → Point {return (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) →Unit {
        print "Foo currently {foo}, now assigned {n}"
        super.foo:= n }

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

class Number {
    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 constructors. The body of an object constructor consists of a sequence of declarations.

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

Object constructors are lexically scoped inside their containing method, or block. In particular, any initializer expressions on fields or constants are executed in that lexical context. (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 constructor is executed, a new object is created.

A constant can be initialized by an object constructor, such as:

```
const unnamedCat := object {
     const color : Color := Color.tabby
     const 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 declarations have no provision for initializing the constant and variable attributes of the created object other than lexical scope. Class declarations combine the definition of an object with the definition of a factory object; the factory object has a method, named new, that creates "instances of the class". A class constructor is similar to an object declaration, except that it may have parameters, like a block:

Examples

```
class Cat { aColour, aName →
    const color  : Color  := aColour
    const 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 parameters to the class body, they are initialized to the arguments to new(). Constants and variables of the result object are inititalized from the parameters and from the class constructor's lexical scope.

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

```
const fergus =Cat.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) to the following nested object declarations:

```
const Cat =object { // the cat factory
    method new(aColor: Color, aName: String) →Cat {
        object { // the cat herself
            const color : Color := aColor
            const name : String := aName
            var miceEaten := 0
        }
    }
}
```

(Option) Classes can have additional factory methods that must return the result of calling new.

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

7.3 Inheritance

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

```
class PedigreeCat implements CatShowExhibit { aColor, aName →
   extends Cat.new(aColor, aName)
   var prizes := 0
   <override> method miceEaten {0};
   <override> method miceEaten:= (n:Number) {return} //Just ignore
}
```

7.4 Understanding Extends (under discussion)

Grace’s class declarations can be understood in terms of a translation to object constructor expressions that build the factory object. Understanding this translation lets expert programmers build more flexible factories. This translation is possible because Grace objects may have more than one field or method of the same name (provided all but the last are annotated <override>).

(Option) Grace provides an **extends** operator that returns an object that is the concatenation of shallow copies of both arguments. The new object has all the fields and methods of each argument, with the left-hand argument taking precedence.

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. If there is just a single argument, the parentheses may be omitted. Parenthesis 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, and the following dot, may be omitted. This is called an implicit method request.

```
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. Grace (currently) does not support variadic methods.

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
```

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 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.

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

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

8.6 Encapsulation

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

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

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 (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) }
```

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

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 {const 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 definition.

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

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

Types describe an object's methods. 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.

Named object or class declarations will define types with the same name as object or class begin defined. The various Cat object and class descriptions (see §7) would produce a type such as the following. (The keyword **method** is omitted as superfluous, since only methods can appear in types.)

```
type Cat {
    color : Color
    name : String
    miceEaten : Number
    miceEaten:=(Number) : Unit
}
```

1. Classes are not types.
2. Classes define a type (of the same name)
3. Somehow, classes need to define a type that describes the objects that are created by their factory methods.
4. Grace will probably use structural subtyping

Acknowledgements

The Scala language specification 2.8 [33] 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 [35, 32], AspectJ

[24], BCPL [36], Beta [28], Blue [25, 26, 27], C [23], C++ [37], C♯ [4, 3], Eiffel [29, 30], Emerald [5], F_1 [10], $F\sharp$ [39], FORTRESS [1], gBeta [13], Haskell [20], Java [12, 16], Kevo [40], Lisp [15], ML [31], Modula-2 [44], Modula-3 [11], Modular Smalltalk [43], Newspeak [8, 6], Pascal [22], Perl [42], Racket [14], Scala [34, 33], Scheme [38], Self [41], Smalltalk [17, 21, 9, 7], Object-Oriented Turing [19] 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 [18].

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. decide about variadic methods §8.1 including Block::apply §5
3. string interpolation syntax §4.4
4. confirm operator precedence §8
5. confirm method lookup algorithm, in particular relation between lexical scope and inheritance §8
6. confirm "super" or other mechaism for requesting overridden methods §8.5
7. confirm rules on named method argument parenthesization §8.1
8. how are (mutually) recursive names initialised?
9. make the **const** keyword optional, or remove it §6.1
10. how should **case** work and how powerful should it be §9.2, see blog post 10/02/2011.
11. support multiple constructors for classes §7.2
12. where should we draw the lines between object constructor expressions/named object declarations, class declarations, and "hand-built" classes? §7.3
13. how do factories etc relate to "uninitialized" §6.2
14. decide what to do about equality operators §10
15. What is the namespace of types? What is the syntax of types? §11 More to the point, what is the type system?

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

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

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

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

[20] 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.

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

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

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

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

[25] 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.

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

[27] 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.

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

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

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

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

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

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

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

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

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

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

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

[39] Don Syme. The F$\sharp$ draft language specification. Technical report, Microsoft, 2009.

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

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

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

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

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