Seeking Grace: A New Object-Oriented Language for Novices

Andrew P. Black
Portland State Univ.
black@cs.pdx.edu

Kim B. Bruce
Pomona College
kim@cs.pomona.edu

Michael Homer
Victoria Univ. of Wellington
mwh@ecs.vuw.ac.nz

James Noble
Victoria Univ. of Wellington
kjx@ecs.vuw.ac.nz

Amy Ruskin
Pomona College
asr02010@pomona.edu

Richard Yannow
Pomona College
rmy02010@mymail.pomona.edu

Grace is the absence of everything that indicates pain or difficulty, hesitation or incongruity.
William Hazlitt

ABSTRACT

Grace is a new object-oriented language that supports a variety of approaches to teaching programming. It integrates accepted new ideas in programming languages into a simple language that allows students and teachers to focus on the essential complexities of programming rather than the accidental complexities of the language. We motivate Grace, review its design, and evaluate it against Kölling's criteria.

Categories and Subject Descriptors

Computing education [**Computer science education**]: CS1; Software notations and Tools [**Language Types**]: Object oriented languages

Keywords

Object-oriented language; novice; CS1; CS2

1. INTRODUCTION

At SIGPLAN's SPLASH conference in 2010, Black, Bruce, and Noble presented a "design manifesto" for a new educational programming language [16], arguing that no existing object-oriented programming language was the obvious choice for teaching novices. Now we are ready to describe the (still incomplete) design for the language, which we call Grace, both to honor the late Admiral Grace Hopper and because we hope that it will enable more graceful programs.

1.1 Why a new language?

Object-oriented programming is the dominant paradigm for commercial software development. At many colleges, it is also the paradigm for the introductory programming courses. Other institutions prefer to start by teaching functional or procedural programming, but even at these places, object-oriented techniques must be taught to relative novices.

What language should be used? Instructors and novice programmers need a conceptually simple language with minimal "accidental" complexity, so that class time can be spent on the key ideas of objects. Today's most widely used teaching languages are overly complex, were designed 20 years ago or more, and often fail to provide good support for features like first-class functions, type parameters, and parallelism.

Instructors preferring statically typed languages have tried C++ and Java, but have not always found success. C++ is a large and complicated language, and novices are often overwhelmed by its complexity. While Java began as a simpler language than C++, over the years it has added features, not always gracefully, and too often requires syntax that cannot be easily explained. Instructors preferring dynamically typed languages have gravitated to Python, but Python has some anomalies as an object-oriented language. Methods must be written with an explicit self parameter that disappears when they are used, and there is no support for information hiding, a key concept in object-oriented programming.

There are other interesting object-oriented languages, like Smalltalk and Eiffel, that are candidates for introductory programming courses. However, they are not now in widespread use in courses, despite being available for many years.

Our focus is on object-oriented programming for general-purpose computing. Thus, while we appreciate the advantages of instructional languages like Karel [14] and Alice [3] that introduce simplified or restricted "computational worlds", we do not wish to restrict our students — typically computer science majors — to programming in a narrow domain.

Much has been written about the choice of languages for introductory computer science courses. Pears *et al.* present a good introduction to this literature in their survey [15, §3.3].

1.2 History of Pedagogical Languages

There is a long history of the use of pedagogical languages in Computer Science. BASIC [10] and Pascal [9] were extremely successful from the 1960's through the 80's, but have fallen out of favor with the rise of new programming paradigms. Other pedagogical languages like Turing [7] have been proposed, but never gained widespread acceptance. Blue [13] was a particularly interesting object-oriented pedagogical language, but had the misfortune of being developed at about the same time as Java, which meant that it did not receive the attention it deserved. Its main legacy is the excellent BlueJ programming environment [12].

One criticism of introducing programming using a pedagogical language is students are less well-equipped to take on internships or other projects with industry. Basic and Pascal overcame this problem to a degree by becoming used in industry; the original Macintosh operating system, for example, was written in an object-oriented extension of Pascal.

We believe that the assumption behind this criticism is that the most difficult and time-consuming part of learning to program is learning the syntax and semantics of a programming language. Instructors who believe this, perhaps teaching at institutions where employment as a programmer immediately after the first course or two is an important goal, will not be interested in using a pedagogical language.

We challenge this assumption. We believe that the most difficult and time-consuming part of learning to program is learning how to solve problems and to represent those solutions as program designs. Once these conceptual tasks are mastered, learning a new language syntax and semantics, at least within the same paradigm, is a straightforward task. Moreover, there are significant advantages to learning programming with a simple and consistent language: students can focus on the important learning tasks without a complex and feature-laden language "getting in the way".

2. GOALS OF GRACE

In this section we list the high-level goals for Grace, along with some specific objectives to reach those goals.

- Grace should integrate proven new ideas in programming languages into a simple object-oriented language.
- The language should gracefully represent the key concepts underlying programming, particularly object-oriented programming, in a way that can be easily explained.
- Grace should allow students to focus on the essential, rather than the accidental, difficulties of programming, problem solving and system modeling.

Let's elaborate on the final point. Brooks, remarking on the difficulties of software technology, divides them "into *essence*, the difficulties inherent in the nature of software, and *accidents*, those difficulties that today attend its production but are not inherent." [1]. We make the same division for programming. The essential difficulties that students must master include concepts like variable binding, conditional execution, iteration, parameter passing, recursion, and information hiding. It is essential to understand these notions to be a good programmer in any language. To these, object-oriented programming adds concepts like dynamic binding and incremental programming (as when using inheritance). In contrast, the accidental difficulties encompass those features of the programming process that have no lasting value, but which must be mastered simply to make progress. For example, to pick on Java, there is no good reason to subject novices to **public static void** *main*(*String* [] *args*) early in a first course, or to have them obsess over which lines should end with semicolons. Thus, a goal of Grace is to eliminate as many of these accidental difficulties as possible, not only because students and instructors have quite enough to do mastering the essential difficulties, but also because we do not want to give students the false impression that programming is about the placement of semicolons.

Some of our specific objectives in achieving these goals are as follows.

- Programs to accomplish simple tasks should be simple, with little or no syntactic overhead.
- Novice programmers should have access to a programming environment (IDE) specifically designed to support novices.
- Language concepts should have simple semantic models.
- Grace should support a variety of approaches to teaching, including:
  - introducing objects first, introducing objects late, and (to a limited extent) starting with functions;
  - an emphasis on static types (like Java), a preference for dynamic types (like Python), and gradual typing, which makes types optional; and
  - a series of "language-levels", so that students may learn progressively in a supportive environment.
- Students who have mastered programming in Grace should find it easy to transition to other languages.

3. AN OVERVIEW OF GRACE

In this section we provide a brief and informal description of Grace, and invite the reader to assess how well we have met the above goals and objectives. More details and examples, including a draft specification, can be found on the Grace website at gracelang.org.

3.1 Declarations and Statements

A Grace program consists of a sequence of declarations and statements, which are executed in the order written. Thus,

```
print "hello world"
```

is a complete Grace program. A more complicated program with the same effect is

```
def greet = "hello "
var subject := "world"
print (greet ++ subject)
```

This illustrates two kinds of declaration: **def** introduces a named constant, which must be given a value when it is introduced; **var** introduces a variable, which can optionally be given a value when it is declared, and can subsequently be assigned a new value. We use = for definitional equality, and := for assignment, as in Algol 60, Pascal, and Eiffel.

3.2 Objects and Classes

Grace classes and objects contain field declarations (both **def**s and **var**s), methods, and code.

```
class aCat.named(n) {
    def name = n
    var livesLeft := 9
    method speak { print "Meow" }
    print "A cat named {n} has been created"
}
var theFirstCat := aCat.named "Timothy"
theFirstCat.speak
```

The class aCat has a constructor called named that takes a single parameter n. When the constructor is invoked by the expression aCat.named "Timothy", an object is created containing three attributes: the constant name, which is bound to the string "Timothy", the variable livesLeft, which is assigned the number 9, and the parameterless method speak. This object is returned as the result of the constructor.

As an effect of creating this object, the string *A cat named Timothy has been created* will be printed; in general, a class can contain arbitrary executable code, which will execute every time an object is created. This example also shows that string literals support interpolation, using a syntax like that of Ruby: the expression inside the braces is evaluated and the resulting object asked to convert itself to a string; this string is then inserted into the quoted text.

We refer to an expression like aCat.named "Timothy" as a "method request", to indicate that an object (here aCat) is being asked to execute a method (here named). There is no need to parenthesize an argument like "Timothy" that is delimited by quotes or braces. Parameterless methods like speak are requested without extraneous parentheses, so Grace programmers write theFirstCat.speak and not theFirstCat.speak().

We use empty parentheses () to indicate the location of a parameter list in a method name, so the above method is more properly called named(). Grace allows multi-part method names (sometimes called mixfix operations), but using a syntax more like Algol than Smalltalk. Thus

```
myVector.at(i)put(newValue)
```

represents a request to myVector to execute the at()put() method with arguments i and newValue.

If only one object of a particular shape is needed there is no need to define a class; the object can be created directly by executing an object constructor. Thus

```
def theSecondCat = object {
   def name = "Timothy"
   var livesLeft := 9
   method speak { print "Meow" }
   print "A cat named {name} has been created"
}
```

results in a object operationally equivalent to theFirstCat.

Everything in the language is an object, including classes, numbers and booleans. Grace does not have a built-in nil or null value. Uninitialized variables are given the special value uninitialized, but this cannot be used as a null reference because attempting to access an uninitialized variable is an error. Instead programmers should create sentinel objects like emptyTree, or use matching as discussed in Section 3.6.

3.3 Types & Information hiding

So far, our examples have not mentioned types, but type annotations are part of the language, as are annotations for information hiding. To support a variety of approaches to teaching, types in Grace programs may be completely omitted, completely supplied, or partially supplied. In all cases, enough dynamic checks are inserted to guarantee that Grace is type-safe.

The default visibility of methods is public. This can be changed by annotating a method as confidential, meaning that it can be requested only of **self**[1] or **super**. In contrast, fields are lexically bound, so are visible only in the object constructor in which they are declared. To make a field accessible to other objects, it may be annotated as readable, which creates a public "reader" method, or writable, which creates a public "writer" method, or both.

Types describe the public interface of objects, like Java interfaces; types can be parameterized by other types, as in List<Number>. Grace types are structural; they are completely separate from classes, and say nothing at all about an object's implementation. So the type Number describes both exact rational numbers and inexact numbers approximated to 64-bit precision, since they have the same methods. This separation of type and class is deliberate. Classes specify how to construct an object, and objects contain the code that gives them their behavior. Types support abstraction and the clear specification of intent, by making explicit the methods that may be requested of an object.

To illustrate the use of types, let's define

```
type Mouse = {
   name −> String
   mass −> Number
   speak −> Done
   increaseMassBy (n:Number) −> Done
}
```

This names a type with four methods — name, mass, speak, and increaseMassBy(). The −> symbol after a method name indicates the return type of that method. The built-in type Done is used to indicate that a method has an effect but does not return an interesting object.

Here is a class whose constructor method named() is annotated to say explicitly that it returns a Mouse object:

```
class aMouse.named(n:String) −> Mouse {
   def name is readable = n
   var mass: Number is readable := 50
   method increaseMassBy (m:Number) {
      mass := mass + m }
   method speak −> Done { print "Squeak" }
}
def jeremy:Mouse = aMouse.named "Jeremy"
```

The type annotation :String indicates that the parameter of the method named() must be a String, and the type annotation −> Mouse indicates that the method returns a Mouse object. The compiler will check that the latter annotation is correct. The returned object has public methods increaseMassBy() and speak, and also methods name and mass generated as a consequence of the **is** readable annotations. The type of name can be inferred to be String because that is the type of n. Because Grace types are structural, jeremy would still be a Mouse even if the declarations of jeremy and aMouse.named contained no type information.

The writer method generated by an annotated declaration

```
var age:Number is writable
```

is called age:=() and is requested using a syntax that is deliberately like assignment: age := 3.

3.4 Inheritance

Grace supports inheritance for both objects and classes. For example we can extend our cat example as follows:

```
class aHipCat.named(n) {
   inherits aCat.named(n)
   def vibe = "Cool"
   method speak is override { print "Meow, man!" }
}
```

The **inherits** clause contains an expression generating a fresh

[1] Grace uses **self** to refer to the current object, corresponding to **this** in Java and C++.

object, typically a request of a constructor method on a class. All of the methods of the inherited object are available on the inheriting object; overriding methods must be flagged with the **is** override annotation.

Because fields are local to the object that defines them, inheriting objects cannot access them directly. However, the annotation confidential, readable, writable will generate reader and writer methods that are restricted to heirs. These methods can be overridden, like any other inherited method.

3.5 Blocks

The Grace *block* represents an anonymous function, otherwise known as a lambda-expression; blocks may be assigned to variables and passed as arguments. A block is written between braces, for example {x −> 2∗x}. The −> separates the parameters of the block from the body; if there are no parameters, the −> is omitted. As is normal in Grace, type annotations on the parameters are optional. The body can be an arbitrary sequence of statements and expressions; the value of the block is that of the last expression executed. Thus, the above example doubles its argument. The body of a block can access identifiers from the surrounding scope, so blocks must be implemented as closures. Blocks are objects; they are evaluated by requesting their apply method:

```
def double = {x −> 2∗x}
print (double.apply(5))          // prints 10
```

Blocks play a central role in Grace, because they enable us to define "control structures" using methods that take blocks as arguments. For example, if()then()else() is a method, defined in a standard library, that takes as arguments a boolean-valued expression and two blocks:

```
if (graceLanguage.isGreat) then {
    print "Adopt it now"
} else {
    print "Come up with a better design"
}
```

The statements in braces are parameterless blocks, which are arguments to if()then()else(). As noted in Section 3.2, parentheses are not required around these arguments because they are delimited by braces. Grace also uses blocks to define internal iterators like map and do over its collections:

```
method averageAge(people) {
    // people is a sequence of Person objects.
    // Compute their average age.
    def ages = people.map{each:Person −> each.age}
    var sum := 0
    ages.do{each:Number −> sum := sum + each}
    sum/ages.size
}
```

This code first uses the map method with the argument block {each:Person −> each.age} to create a sequence of ages, and then the do method with the argument block {each:Number −> sum := sum + each} to compute their sum.

A method returns the value of the last expression evaluated, which in the method above is sum/ages.size. An explicit **return** statement can be used to specify an early return; this is useful inside a block, because **return** means return from the enclosing method, *not* from the block itself. For example, the following method might be defined on a collection that has a do() method for iteration.

```
method detect(p:PredicateBlock)ifNone(a:ActionBlock) {
    // find element in this collection that
    // satisfies predicate p. If none, execute a.
    self.do{each −>
        if (p.apply(each)) then {return each}
    }
    a.apply
}
```

As in many other object-oriented languages, the receiver **self** can be omitted.

Grace's use of parentheses and braces is not arbitrary. The expression used as the test in if()then()else() is evaluated once when the method is requested, so it is parenthesized. In contrast, because the code following the then and else parameters must be evaluated conditionally, those expressions must be blocks. While this is different from some other languages, it provides an opportunity for instructors to discuss conditional and repeated execution of code.

A benefit of first-class blocks is that library writers, instructors, and students can write methods that provide high-level operations on their data objects. Being able to write and reuse common abstractions like the detect()ifNone() method, rather than requiring that clients "roll their own" using an iterator and a loop, is something that we consider an important part of object-oriented programming.

3.6 Pattern matching

Grace provides pattern matching similar to that found in languages like ML, Haskell, and Scala. Unlike these languages, Grace's pattern matching facilities are mostly definable using the core features of Grace, and hence can be supported through libraries. A complete description is the subject of another paper [8]; here we confine ourselves to a brief sketch.

The programmer can use pattern matching on both literal values (like the switch statements in the C family of languages) and on types. For example:

```
match (exp)
    case { 0 −> "Zero" }
    case { n:Number −> "Number less than {n+1}" }
    case { s:String −> "String \"{s}\"" }
```

This expression first evaluates exp to yield an object, and then selects a string describing that object.

Pattern matching can also be used to extract information from an object, but only if the object provides a method to make that information available. For example, suppose that we wish to sum the elements of a list of numbers

```
type List<T> = {
    head −> T
    tail −> List<T>
    isEmpty −> Boolean
    extract −> Tuple<T,List<T>>
    ...
}
method sum(aList:List<Number>) −> Number {
    match(aList)
      case {(emptyList) −> 0}
      case {xs:List<Number> −> xs.head + sum(xs.tail)}
}
```

The method sum first matches aList against the constant emptyList. If this fails, aList must have a head and a tail, and the method can perform the obvious recursion. However, because List also provides the method extract, sum can also be written using a so-called destructuring match:

```
method sum(aList:List<Number>) −> Number {
   match(aList)
      case {(emptyList) −> 0}
      case {List<Number>(hd, tl) −> hd + sum(tl)}
}
```

Here, hd and tl are not arguments, but sub-patterns that bind variables. They are bound to the values returned by the extract method on aList. Destructuring matches can be nested arbitrarily deeply.

Patterns are just objects that understand a particular protocol. Consequently, in addition to the built-in patterns described so far, users and library writers can define their own patterns. For example, it is possible to provide a library that supports the matching of strings against regular expressions.

It is useful, particularly for novices, if the compiler can provide a static warning when a pattern match is not exhaustive, and when cases are unreachable. To permit this, Grace includes tagless *variant types* of the form T | U. To have the type T | U, an object must have either type T or type U. The combination of variant types and *singleton types* make it straightforward to program without nil. For example, the object emptyList above, which does not have all of the methods of type List, has the type singleton(emptyList); we call this a singleton type because emptyList is the only object with this type. We could then declare

```
type ListOption<T> = List<T> | singleton(emptyList)
```

Methods operating on a ListOption object can use pattern matching, or explicit emptiness tests, to deal with the two cases. The advantage of pattern matching is that in the body of the case {xs:List −> ...}, xs is statically known to be non-empty.

4. THE STATE OF GRACE

Most of the details of the language design have now been worked out, though we expect refinements to be made as we gather experience with the language. We have written a substantial amount of Grace code, including a self-hosting compiler for Grace itself. The compiler is called *minigrace*; it generates either JavaScript, which can be run in a web browser, or C, which can run on any platform with a C compiler. We have also written parts of a collections library for Grace, and a selection of typical assignments for a data structures course. All code can be reached from the Grace language web site at gracelang.org.

We are convinced that the success of a language for novices depends on having a supportive interactive development environment (IDE). Rather than designing such an environment from scratch, we have examined the feasibility of modifying BlueJ and DrRacket (formerly DrScheme) to support Grace.

We currently have a partial implementation of Grace inside of DrRacket [5], and intend to complete that implementation and to create language levels for Grace in DrRacket. We anticipate having a directed graph of languages rather than a linear progression so that we can support multiple approaches to teaching object-oriented programming.

A BlueJ implementation requires an implementation of Grace on the JVM. While we would eventually like to build such an implementation, it is currently on a back burner.

There is a wide consensus that teaching parallel programming is of growing importance. Unfortunately, there is no consensus on what kind of parallelism to teach: message-passing, shared variables, futures, actors, or whatever. To avoid committing to a model that will be wrong for a large fraction of our intended audience, Grace will provide a variety of libraries for supporting parallelism in a variety of styles. We don't believe that such support need be "second class", because most of the basic features of Grace, including all of the control structures, are also provided by libraries.

We have had good experience with an undergraduate assistant writing libraries for data structures, but must develop tool support and teaching materials to class-test Grace. We plan on using Grace in small classes of novices in late 2013. We expect to make revisions to Grace as we respond to any learning difficulties we find in these early classes.

We intend to develop at least two different approaches to teaching with Grace, one based on the text by Felleisen *et al.* [4], which emphasizes "design recipes", and one based on the text by Bruce *et al.* [2], which takes an objects-first approach. We will make the lecture notes and examples from these courses freely available online.

5. EVALUATING GRACE

While it is difficult to evaluate Grace without direct teaching experience, we can provide a first evaluation based on criteria that others have presented on language choices for novices. For this purpose we adopt the checklist of eleven criteria proposed by Michael Kölling [11].

1. **Clean concepts.** We have provided uncluttered features with a straight-forward semantics. Students can begin by defining objects directly before being introduced to classes, and simple objects can be created fully formed, without need for complex initialization. Types and classes are separate concepts: classes are used to create objects, while types are used to specify the interface of an object, without reference to any particular implementation.
2. **Pure object-orientation.** Everything in Grace is an object, including numbers, booleans and blocks. As a result, there is no need for the complications of value versus reference. While Grace supports functional constructs, they are obtained in an object-oriented way.
3. **Safety.** Grace is strongly typed, with type errors caught at either compile time or run time. Because types are gradual, students may write programs across a spectrum from no type annotations, resulting in dynamic checks, to complete annotations, resulting in static checks. We have designed Grace to make it possible to check for access to uninitialized variables, exhaustiveness of type-case matches, and other common errors.
4. **High level.** Grace is high-level in several ways. Of course, Grace uses automatically managed memory, eliminating the possibility of some of the most frustrating errors for students. More importantly, blocks are first-class in Grace; this provides students (and teachers) with the ability to write methods that accept blocks as parameters, which in turn makes it easy to treat data structures as "whole objects". A good example is a method that traverses a tree, searching for a value that satisfies a predicate represented as a boolean-valued block.
5. **Simple object/execution model.** All objects are on the heap, and all computation takes place by requesting that an object execute a method.
6. **Readable syntax.** All declarations are introduced by

keywords; we chose keywords that directly signify the concept that we wish to teach. So, for example, methods are introduced by the **method** keyword, not **fun** or **mdef**.

Because we expect students to transition to languages that use C-style syntax, we have adopted features that are close to those languages, though normally with a twist. For example, we use braces to delimit blocks, but those blocks must *also* be consistently indented. Semicolons can always be omitted at the end of a line.

In declarations, names come before types, making the names more visible. We distinguish between "=" for definitions and ":=" for assignment to variables because they represent different concepts.

7. **No redundancy.** This is difficult to verify, but we have worked to build Grace from a small number of primitive constructs. Because blocks are first class, we can build any number of looping constructs. That might be considered redundant, but we feel it is important to be able to use the most appropriate loop for a given traversal.

   Pattern-matching may seem to be redundant: most object-oriented languages don't have it, and some programmers feel that pattern-matching is inherently non-object-oriented. Most languages *do* provide a construct to check the type of a run-time value; pattern matching provides similar facilities, and in a way that facilitates static type-checking. Moreover, pattern matching can be excluded from the beginning language levels.

   However, the real argument for including pattern matching in Grace is that it makes Grace *a more effective teaching language*. Students should be able to compare code written with and without pattern matching, and learn from experience which is more readable and easier to maintain. This comparison is most effective if all other variables can be help constant, which means comparing two equivalent programs written in the same language.

8. **Small.** The core of Grace is quite small; we avoided enlarging the language with anything that can be defined in a library. Grace does include a simple module facility for importing features from libraries, but does not provide many facilities for programming-in-the-large, as would be needed in an industrial-strength language. Students should be able to learn all of Grace in one or two semesters.

9. **Easy transition.** The basic concepts supported by Grace are available in other object-oriented languages, though often obscured by complicated syntax. Even blocks, not currently part of Java, will be added in the next revision. They are already included in C# 3.0 [6].

10. **Correctness assurance.** We do not currently have annotations for pre- and post-condition, variants and invariants, but we expect to provide them in libraries.

11. **Environment.** We are well aware of the importance of a good programming environment for Grace. As described in Section 4, development of an environment is underway.

We believe that Grace stacks up very well against languages like Java, C#, Python, and Eiffel with respect to Kölling's criteria. We are well aware that a positive reception by students is crucial to the success of the language. Thus, formative evaluations with real classes of students will be essential to the further development of the language.

6. CONCLUSION

While this project is still in relatively early stages, we feel that it is now far enough along to present to potential adopters. We are eager to have feedback on the language design to this point, and are happy to have others pitch in with library designs, work on programming environments, improvements in the implementation, and in testing the language on novices.

More information on the project, including information on the current prototype compiler, is available at gracelang.org.

7. REFERENCES

[1] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[2] K. Bruce, A. Danyluk, and T. Murtagh. *Java: An Eventful Approach.* Prentice-Hall, 2005.

[3] W. P. Dann, S. Cooper, and R. Pausch. *Learning to Program with Alice.* Pearson Prentice Hall, 2011.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to design programs: an introduction to programming and computing.* MIT Press, Cambridge, MA, USA, 2001.

[5] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[6] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *C# Programming Language.* Addison-Wesley, 4th edition, 2010.

[7] R. C. Holt and J. R. Cordy. The Turing programming language. *Commun. ACM*, 31(12):1410–1423, 1988.

[8] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *Proc. 8th symp. on Dynamic languages*, DLS '12, pages 17–28, New York, NY, USA, 2012. ACM.

[9] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer, 1975.

[10] J. G. Kemeny and T. E. Kurtz. *Back to Basic; The History, Corruption, and Future of the Language.* Addison-Wesley, 1985.

[11] M. Kölling. The problem of teaching object-oriented programming, Part I: Languages. *JOOP*, 11(8):8–15, 1999.

[12] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), Dec 2003.

[13] M. Kölling and J. Rosenberg. Blue — a language for teaching object-oriented programming. In *Proc. 27th SIGCSE technical symp. on Computer science education*, SIGCSE '96, pages 190–194, 1996.

[14] R. E. Pattis. *Karel The Robot: A Gentle Introduction to the Art of Programming.* Wiley, 1995.

[15] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '07, pages 204–223. ACM, 2007.

[16] J. Rosenberger. Grace: A manifesto for a new educational object-oriented programming language. Blog@CACM, October 2010. Retrieved Dec 2012. http://cacm.acm.org/blogs/blog-cacm/100389.