# The Importance of Being Eelco

**Andrew P. Black**
Portland, Oregon, USA
**Kim B. Bruce**
Pomona, California, USA
**James Noble**
Creative Research & Programing, Wellington, NZ

## Abstract

Programming Language Designers and Implementers are taught that:
semantics are more worthwhile than syntax,
that programs exist to embody proofs, rather than to get work done,
to value Dijkstra more than van Wijngaarden.
Eelco Visser believed that, while there is value in the items on the left, there is at least as much value in the items on the right. This short paper explores how Eelco Visser embodied these values, and how he encouraged our work on the Grace programming language.

**2012 ACM Subject Classification** Software and its engineering → Syntax; Software and its engineering → Semantics; Software and its engineering → Object oriented languages; Software and its engineering → Translator writing systems and compiler generators

**Keywords and phrases** Eelco Visser, Grace, syntax

**Digital Object Identifier** 10.4230/OASIcs.CVIT.2016.23

**Funding** *James Noble*: This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden, and Agoric.

## 1 Introduction

2010 was a long, long time ago, if not in a galaxy far, far away. Barack Obama had been president for a couple of years, Boris Johnson was onto only his second wife, and the idea of Donald Trump as a political figure was far from the mind of even the most fevered cartoonist.

The programming languages landscape appeared moribund — especially for academics who were teaching introductory programming courses. According to the TIOBE index [31], Java, C, and C++ were the three most popular languages between 2007 and 2017. Python was slowly bubbling up (from 8 in 2007 to 5 in 2017), and the PLT Scheme project was just starting the process of changing the name of their language to Racket [1].

Adopted with alacrity during the first decade of the 21$^{st}$, Java had become a *lingua franca* for teaching and research, and had been widely adopted in industry [38]. In 1998, Java 2 was a relatively small language, with a fairly conventional syntax, and a collections library. By 2010, Java 5 — the then-most-recent major version of Java, which introduced generics — had been available for six years. The next major Java version, Java 8 — which supported lambdas and streams — was still four years in the future.

This was the context in which the three authors embarked on the Grace project, and in which Eelco Visser embarked on the Spoofax Project.

### 1.1 A Short History of Grace

As teachers and academic researchers, we were faced with finding a language for our teaching and research [14, 17, 24, 30, 37]. For teachers, such a choice should be pedagogical: how many

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:14
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

languages, what kinds of languages, and to what level of expertise should we expect computer science or software engineering graduates to know? For programming language designers, the choice is aesthetic: what values or design philosophy does a programming language embody, and are they the values and philosophy we hope to impart to our students? Pragmatic considerations are also in play: while high-prestige institutions can teach and conduct research in more-or-less whatever language they like, smaller or less prestigious institutions have more limited expertise, time, and resources, and every additional programming language imposes a cost. For researchers, students and teachers, programming languages are human languages [26]: they are used to communicate between people as much if not more so than to communicate between people and machines. We use programming languages to share and explain *ideas*. There are practical limits on how many languages people are able to implement, to understand, or even to parse. The size and complexity of a language also matters: there are limits to how much of a language can be taught in a one-term or one-semester course (both a first course, and a course used to introduce a new language later). In 2010, Java 5 seemed to be pushing those limits, if it had not already exceeded them. Finally, the choice of language is also ideological: should it be procedural, functional, or object-oriented? Should it be statically typed or dynamically typed, or something in between? Should it be designed for its purpose, or chosen from a menu of “industrial strength” languages?

The question of which language to use for teaching came to a head at ECOOP 2010 in Maribor. In hallway conversations, we asked ourselves, as language designers and implementors, whether or not we should be working on a language targeted at our own needs? After all, developing languages was becoming easier, thanks to common runtime environments like the JVM and CLR, and IDEs and language workbenches like Eclipse, JetBrains MPS, and the PLT Scheme/Racket tooling. It seemed (after a beer or too) that, just as Haskell, and before that Algol, had been built by teams of academics, it might be possible to design and implement a new language suitable for teaching, and usable as a base for research, as a neutral cooperative effort amongst academics and not tied to particular companies or projects.

After an initial flurry of interest sparked by a “Manifesto” published at SPLASH 2010 [4], the task of designing Grace was taken on by Black, Bruce and Noble, the authors of the present paper. We met weekly in cyberspace, and less frequently in person. We also presented progress reports and requests for feedback at IFIP WG2.16, and at workshops organized in conjunction with major programming conferences.

## 1.2 A Short History of Spoofax

The initial version of Spoofax was designed by Kats and Visser as a language workbench for simplifying the development of new domain-specific languages. Implemented as plug-ins to Eclipse, Spoofax integrated a variety of tools: SDF2 [36] for specifying grammars, generators of customizable editor service descriptors based on the syntax, and (initially) the Stratego program transformation language [6] to describe semantics using re-write rules. These individual tools had been long in gestation; Stratego in particular went back to Eelco’s time as a postdoc at the Oregon Graduate Insitute in the late 1990s.

Spoofax evolved to encompass additional tools, in particular the DynSem [35] language for specifying dynamic semantics. Behind all of these tools was Eelco’s drive to bring the power of modern computers to the task of language implementation. Twenty-first century programmers, for example, have come to expect an IDE with a robust set of language-specific editor services. Without such tooling, a new language, whether domain-specific or general-purpose, would struggle to gain a toehold. Eelco saw that by capturing a language design in a series of DSLs, much of the gunt work of producing an implementation and he associated toolsing could be automated.

That, at least, was his vision (from our perspective, as outsiders). Could it be realized?

# 2 A Little Grace

Our subject here is Eelco as much as Grace, so we will limit ourselves to describing the features of Grace relevant to our collaboration with Eelco and his group. We refer those interested in a more complete description of Grace as it stood around the time relevant to this article to the short papers presented at SIGSCE [8] or IEEE CSEE&T [28].

## 2.1 Objects and Classes

A Grace object is created by executing an object constructor, which is a special kind of expression introduce by the reserved word **object**. Each time an object constructor expression is evaluated, a new object is created and returned. Here is an example:

```
object {
    def name = "Fido"
    var age := 0
    method say(phrase : String) {
        print "{name} says {phrase}"
    }
    print "{name} has been born."
}
```

The object created by executing this constructor contains a method say and two fields; **def** name defines a constant (using =), while **var** age declares a variable, whose initial value is assigned with :=. New values can be assigned to variables, also with :=. When an object constructor is executed, any code inside its body is also executed, so the above object constructor will have the side effect of printing “Fido has been born.” when the object is created. This example also shows that strings can include expressions enclosed in braces: the expression is evaluated, converted to a string, and inserted in place of the brace expression.

Of course, to be useful, the object created by executing an object constructor typically needs to be bound to an identifier, or returned from an enclosing method. For example,

```
method dog(n:String) {
  object {
    def name is public = n
    var age is public := 0
    method say(phrase : String) {
        print "{name} says {phrase}"
    }
    print "{name} has been born."
  }
}

def fido = dog "Fido"
fido.say "Hello"
```

will create an object and bind it to the name fido, and then *request* the say method on that object. The constructor will print "Fido was born." and then the request of the say method will print "Fido says Hello". Grace uses the term "method request" in preference to "message send", because "sending a message" might be misinterpreted as referring to a network message. We prefer "request" over "call" to recognise that the receiver must cooperate in responding to the request.

The construction in the above example — a method whose body is an object constructor — plays the same role as a class in a language like Python: it creates an object that can be parameterised by the arguments to the method (here, the name of the dog). Grace has a **class** construct to make this more convenient, but classes are second-class: **class** is nothing but a shorthand for a method that returns a freshly-constructed object. The code below is exactly equivalent to the code above.

```
class dog(n:String) {
    def name is public = n
    var age is public := 0
    method say(phrase : String) {
        print "{name} says {phrase}"
    }
    print "{name} has been born."

}

def fido = dog "Fido"
fido.say "Hello"
```

Grace also has a **trait** keyword, which is similar in function to **class**: it defines a method that returns a freshly-constructed object. The difference between **trait** and **class** is that the object created by a **trait** may not contain any fields. The purpose of a trait object is to package-up a bundle of methods so that they can be reused in another object.

## 2.2 Syntax and Layout

As you can see from these examples, Grace's syntax is a relatively conventional mix of the "curly bracket" style of C and the keyword style of Pascal. Declarations and code blocks are delimited by `{...}` rather than `begin...end`, but declarations are marked by keywords (e.g., **def**, **var**, **method**). We hoped that this would make the syntax clearer to novices, as well as teaching them important vocabulary. Types follow identifiers after a colon, and assignment is := rather than =, so Grace writes **var** x:Number := 52 rather than int x = 52. This makes it possible to omit types entirely if that is what the instructor prefers. Control structures intersperse keywords between the components: "if (flag) then { }" rather than "if (flag) { }". Control structures are not built in; instead they are methods that use Grace's multiple-part method names.

It seemed unnecessary and ugly to require parentheses around a block that is already delimited by braces, or around a string that is already delimited by quotes. Consequently, many request arguments don't need to be parenthesized; arguments are enclosed in parentheses only when necessary to avoid ambiguity or to promote readability.

As well as using braces to indicate the boundaries of code blocks and declarations, Grace requires that code layout must be consistent with these boundaries. That is, indentation must increase after an opening brace, and return to the prior level with (or after) the matching closing brace. Statements may be separated by line breaks or by semicolons:

```
def x =                              def x =
    mumble "3"                           mumble "3"
    fratz 7                              fratz 7;
while {stream.hasNext} do {          while {stream.hasNext} do {
    print(stream.read)                   print(stream.read)
}                                    };
```

**Andrew** ►*The above example relies on the rule that indentation indicates a continuation line, which we don't explain until 2 paragraphs further on*◄ **Kim** ►*If we leave in the second example above, put in all semicolons to make the idea clearer.*◄

Indentation is not purely a matter of consistency. It is also used to distinguish between a single request of a method with a multi-part name, and multiple requests of methods with single part names. Consider Grace's "if(_)then(_)else(_)" control structure. The body of the code block that forms the argument for the then part should be indented more than the line that contains the opening {, and the closing } is at the same indentation as the line that contains the opening {. (See the left column below.) Because there is no line break after the first }, the else(_) cannot be a separate method request.

Because indentation is also used to indicate a continuation line, an alternative format for our example is to indent the then and the else, in which case the whole if(_)then(_)else(_) will be treated as a single logical line, as shown in the center column below. This format is appropriate only when the code blocks are small.

A consequence of these rules is that lining everything up on a common left margin is not a valid way of formatting a single if(_)then(_)else(_): such a layout will be interpreted as three separate method requests: an if(_), a then(_), and an else(_), shown in the right column.

```
if (condition) then {
    doThis                if (condition)               if (condition)
} else {                      then { doThis }          then { doThis }
    doThat                    else { doThat }          else { doThat }
}                         theFollowingStatement        // three separate requests
theFollowingStatement
```

## 2.3 Nesting and Inheritance

As in most object-oriented languages, Grace objects and classes can inherit from one another. For example, we can define a simple object out that inherits from a SuperClass:

```
class superClass {
    method m { "in superclass." }
}

def out = object {
    inherit superClass
  method foo { print (m) }
}

out.foo
```

Grace follows Java and many recent languages in allowing the programmer to elide **self** in method requests. The `m` in `print(m)` in **method** `foo` is actually shorthand for **self**.`m`. Notice how method `m` must be inherited by object `out` for this code to work.

Grace supports reuse in two ways: through **inherit** statements and through **use** statements. Classes can reuse the attributes of a single superclass via an **inherit** statement, and can reuse the methods bundled in multiple traits via **use** statements — this form of multiple inheritance is benign because traits are stateless. Method renaming and method exclusion is permitted for both kinds of reuse [27].

Grace's objects — like those in most contemporary object-oriented languages, arguably going back to **Andrew** ▶*Simula 67 ? and* ◀BETA [25] — also can be lexically nested. For example, we could define a second object `inner` that is lexically inside the object `out`:

```
def out = object {
        method m { "in enclosing object." }
    def inner is public = object {
        method foo { print (m) }
    }
}

out.inner.foo
```

Now the method `foo` is inside *two* objects: the object `inner` and the object `out`. What then is the meaning of the unqualified `m` in `print(m)`? We can see that it cannot mean **self**.`m` because **self** — the object `inner` — does not have an `m`. We deduce that it must mean **outer**.`m`, that is, the `m` defined in the object lexically surrounding **self**.

The devil is always in the details, or rather the ordure is in the orthogonality. We have adopted three features, seemingly orthogonal, and seemingly useful: inheritance, nesting, and implicit **self**. What if a program attempts to use all three mechanisms at the same time?

```
class superClass {
        method m { "in superclass." }
}

def out = object {
        method m { "in enclosing object." }

    def inner is public = object {
            inherit superClass
        method foo { print (m) }
    }
}

out.inner.foo
```

Which `m` does the method `foo` invoke: the `m` in the lexically-enclosing object **outer**, or the `m` inherited from `superclass`?

This potential ambiguity is common across many object-oriented languages — with as many different solutions as there are languages [5]. Java uses "up then out" semantics, and thus would invoke `m` inherited from the superclass. Newspeak uses "out then up", so would invoke `m` in the enclosing object. As a language designed for education, Grace simply bans such ambiguous requests, requiring that the programmer resolve the ambiguity by writing **self**.m or **outer**.m [27].

As Tony Hoare explained almost fifty years ago:

> The principles of modularity, or orthogonality, insofar as they contribute to overall simplicity, are an excellent means to an end; but as a substitute for simplicity they are very questionable. [21, p.7]

The issue is not just simplicity *vs.* orthogonality, but rather where and when does complexity appear, and whether orthogonality increases simplicity, incubates complexity, or both.

## 3 Grace in Spoofax

The original goal of the Grace project was to produce a language specification, not a language implementation [4, 3, 9]. While at least one implementation would be essential even to guide the process of writing the specification, we hewed to the 20th century ideal that a programming language should be implementation independent.[1] Build it (the specification), we thought, and they (the implementors) will come. How naïve we were!

### 3.1 SDF2 Grace Parser

But come they did, or rather, Eelco Visser and his Spoofax team came: notably Master's student and doctoral students Vlad Vergu and Luis Eduardo de Souza Amorim. We recall Eelco attending, slightly bemused, the meeting at ECOOP 2010 where the Grace project was mooted. He was too wise to sign on to the SPLASH 2010 "Manifesto" [4], but as one of the early forces behind IFIP WG2.16 he was certainly aware of the Grace design effort. By the time of the first official meeting of WG2.16 (in London, in February–March 2012), the first iteration of Grace's design was complete. In an email exchange **Andrew** ►*with whom?*◄ following up on a "conversation after the pub" Eelco was interested in becoming an early implementor, going so far as to say:

> Rather than farming this out to a student, I'm planning to make it my 'trying out new features of Spoofax and learning about design choices in (OO) language design' project, with all the risks associated with that, so don't hold your breath.

Eelco was a good as his word. Working off an early grammar for Grace (at the time, self hosted via a parser combinator library within Grace itself), by OOPSLA at the end of the 2012, Eelco had the bones of a Spoofax parser working for Grace. Somehow, he had written this in his spare time! The Spoofax parser could handle essentially all the language as defined at the time, with the exception of Grace's then ill-defined layout syntax rules: rather than relying on indentation and line breaks, Spoofax-Grace statements had to be terminated with semicolons, and there was no enforcement of Grace's requirement that indentation be consistent with brace-structure.

---

[1] How wrong we were: pretty much every successful programming language since has been based on a single canonical implementation.

## 3.2 Spoofax–Grace

Eelco's parser was then extended by Michiel Haisma for his Master's thesis, resulting in a fairly complete implementation of the core of Grace completely within the Spoofax environment. (One of our initial goals for the Grace project was that the language should be implementable by a couple of graduate students in about a year: Haisma's thesis demonstrates that this goal could be met by talented students using the right tools).

Up to this point, Grace's specification was informal, and existing implementations were hand-coded interpreters and compilers. The aim of Spoofax–Grace was not just to provide an implementation of the Grace programming language, but also to serve as a reference implementation that could be tested, and as a specification that could be easily read, understood and changed [18, 34, 19].

Figure 1 shows the architecture of Spoofax–Grace. Spoofax's SDF3 DSL [12] parses Grace code into an initial AST. Next, the Stratego transformation language rewrites some Grace constructs (such as classes and traits) that are actually defined in terms of simpler constructs (such as methods and objects) in a "desuguring" pass. A "lowering" pass then produces a canonical, fully decorated AST [7]. Finally, definitions in the DynSem DSL [35] are used to actually execute (i.e., interpret) the program represented by the lowered AST.

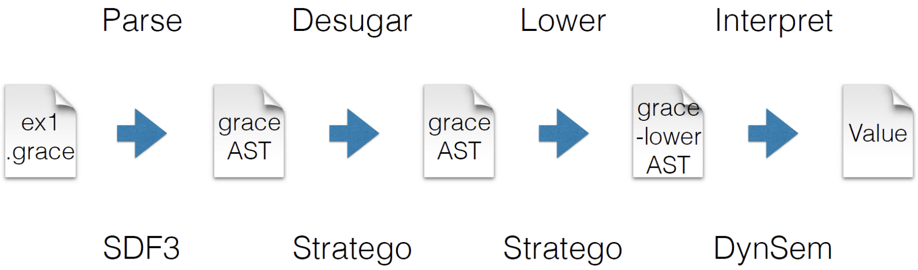
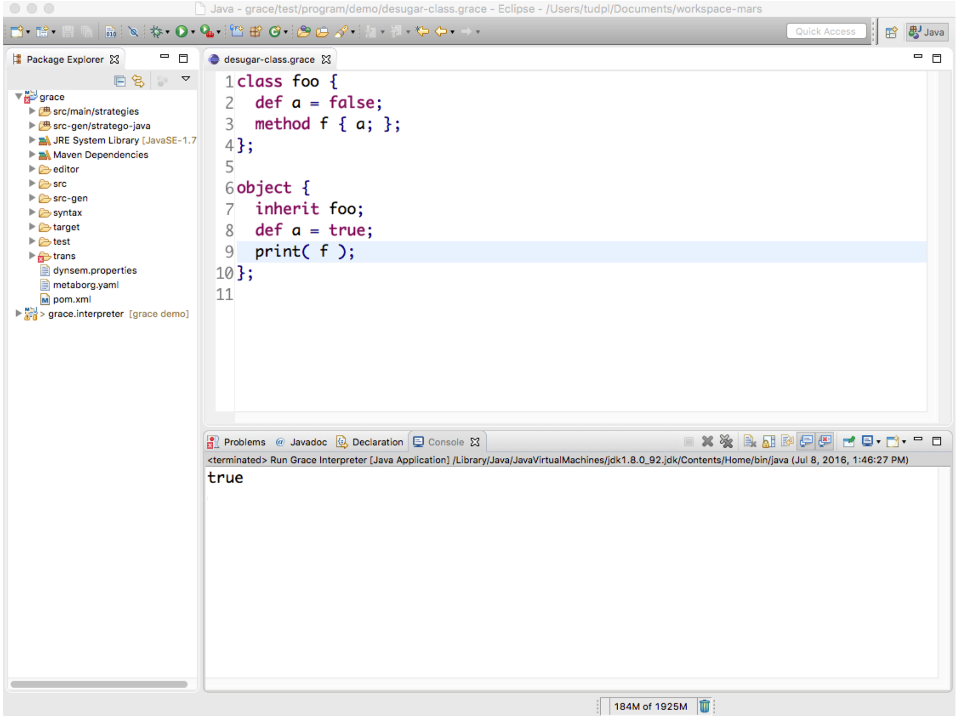


**Figure 1** Spoofax–Grace Architecture. From Michiel Haisma, "Grace in Spoofax" [18], used with permission

Figure 2 shows the system running a simple Grace program inside Eclipse. The leftmost column contains the Eclipse explorer; the central window shows the Grace source code — pretty-printed and syntax-coloured automatically from the Spoofax definitions. The bottom window shows the output of the DynSem interpreter executing the Grace program — here, simply: `true`.

The Spoofax–Grace project illuminated some details of the Grace specification as it existed at the time, and made us realize that the specification was not as precise as we had thought. One important area was the semantics of request resolution: the part of the language that had to combine the local definitions within an object, definitions inherited from superclasses, reused from traits, located in enclosing lexical scopes, or from the module's dialect or prelude [22, 27]. Based upon Eelco's theory of Scope Graphs [29, 32], Grace's lookup semantics were encoded using the Spoofax DynSem DSL [35], as part of the overall operational semantics. The DynSem implementation was shorter and easier to modify than the existing Grace implementations [34], and also clarified that the computational complexity of a Grace method request in an object $n$ nested levels deep, with $p$ parent objects (traits and superclasses), was $O(np)$.

The Spoofax implementation of Grace was actually more general than we, as the designers of Grace, had ever intended. Because we had always planned for Grace to have a static



```
class foo {
  def a = false;
  method f { a; };
};

object {
  inherit foo;
  def a = true;
  print( f );
};
```

```
<terminated> Run Grace Interpreter [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Jul 8, 2016, 1:46:27 PM)
true
```

**Figure 2** Grace in Spoofax. Michiel Haisma, used with permission.

type system, it was important throughout the design that the *shape* of a Grace object — the methods and fields that it contained — could be determined statically. Although **inherit** and **use** statements describe the parent object (the object being reused) with *expressions*, we intended that these expressions be *manifest*, that is, evaluable statically. But the specification document didn't make this clear enough, and Vergu *et al.* write: "The use of expressions to determine ancestors means that meaningful name resolution can only be performed at run time" [34]. The Spoofax implementation agreed with our prototype implementation in the sense that any Grace program that was accepted by the prototype gave the same results in Spoofax, but the Spoofax–Grace implementation allowed for reuse of objects whose shapes could not be ascertained until run time. This experience was enlightening to all involved, and made it clear to us that we needed to do some serious work on our language specification. In particular, we needed to overhaul the definition of *manifest*.

The purpose of the Spoofax–Grace project was as much to evaluate the Spoofax toolset as the Grace specification. Other than the difficulty of handling layout, the toolset performed admirably: deficiencies of Grace-Spoofax (missing pattern matching, lack of a type system and static analyses) were due more to a lack of time, or to imprecision in the language specification, than to weaknesses in the tools. At the time, Spoofax was also competitive with the other Grace implementations in the time required to make a small change to the definition and rebuild the system (see fig. 3). **Andrew** ►*If this is important, we need to explain the numbers. Or, we could take it out.*◄

Looking back on this episode, one lesson that could be drawn is that a clear separation between static and dynamic semantics might have been beneficial to both language designers and to Spoofax users. There are places where the Grace specification deliberately leaves open the question of whether a check is static or dynamic, to allow the implementor more freedom. However, this should be done explicitly, as is done for, for example, type checks " The checks necessary to implement this guarantee [type safety] may be performed statically or dynamically", and not by obscure phrasing or by omission. Another lesson is the value of the Agile practice of the on-site customer [2]: if the Grace and Spoofax teams had been co-located, this lack of clarity about what could be inherited would have been discovered much sooner.

The Spoofax implementation of Grace is available [20], although not currently being maintained. It now also includes a version of a parser that can handle Grace's layout, based on extensions to SDF3 to support indentation which were made while the main Spoofax–Grace project was coming to an end [12, 13].

| **Implementation** | **Time (initial) (s)** | **Time (change lexer) (s)** | **Time (change semantics) (s)** |
|---|---|---|---|
| **Spoofax** | 91 | 91 | 49 |
| **Hopper** | 0 | 0 | 0 |
| **Kernan** | 2,5 | 2,5 | 1,3 |
| **Minigrace** | 163 | 10 | 14 |

**Figure 3** Compile time. From Michiel Haisma, "Grace in Spoofax" [18], used with permission.

## 4 The Eelco Manifesto

In the abstract, we made some presumptuous claims. If Eelco were still with us, we would do so cavalierly, knowing that Eelco would take our comments in good heart, and enjoy disputing with us. Sadly, that will not happen, so we proceed with more caution. We will do our best to justify these claims, and leave it to posterity to decide if they have value.

### 4.1 Semantics and Syntax

We are going to say it outright: syntax is important! Yes, semantics is important too, but the semantics has to be attached to something: syntax *carries* the semantics.

In Spoofax, Eelco acknowledged the place of syntax. Parsing, pretty-printing, and editor support are important to the programmer. They are, or ought to be, the cornerstone of any language implementation. It is certainly possible to produce a language workbench that ignores syntax — the input language could be S-expressions — and focuses instead on semantics, optimizations, and execution. But that would have set aside a lot of what concerns users, and abdicated responsibility to help implementors in an area where tooling is both important and effective.

### 4.2 Program Proofs *vs.* Working Code

Looking through the proceedings of computer science conferences, where one used to find descriptions of working programming *systems*, one now finds descriptions of formal calculi — Featherweight Java, System F, and so on. One can see traces of this trend as far back as 1979, when Dijkstra thought it appropriate to ridicule Teitelman's Interlisp system because the "reference manual for Interlisp is already something like a two-inch thick telephone directory" [16]. Having an extensive library was apparently a fatal flaw in Dijkstra's eye.

Yes, there is value in formal calculi, and there is value in proofs of correctness. There is also value in complexity theory and in choosing an appropriate algorithm. But the *reason* that these things have value is because programs do stuff in the real world, and we want them to do the *right* stuff, and we want it done quickly.

Eelco, as exemplified in Spoofax, was interested in a system that worked in the real world — for example, that integrated with Eclipse, and provided programmers with editing tools that were satisfying to use. Yes, the Spoofax tools were built on sound theoretical foundations. But foundations alone were not enough: they had to get work done.

As language designers, we appreciate the value of formal systems. When one changes one's grammar, it's nice to know that the grammar remains unambiguous. When one changes one's type system, it's nice to know that the type system remains sound. But there is also enormous value in having a working implementation on which you can run examples. We can remember occasions where, after long discussions and some longer walks, we agreed on a change to Grace. Then one of us started programming in the revised language, and was forced to confront the consequences of the change! Of course, if we had only been smarter, we could perhaps have foreseen these consequences. Alas, we are who we are. Having an implementation that could quickly show us the consequences of a change, and show it on a sizable body of code, was of enormous value during the design process.

## 4.3 Dijkstra and van Wijngaarden

Although Adriaan van Wijngaarden was Edsger Dijkstra's boss and academic supervisor, the two men could hardly have been more different. Let us concentrate here on two differences. First, where van Wijngaarden was an enthusiastic adopter of new technology, Dijkstra didn't seem interested.

This seems to be an odd comment to make about one of the pioneers of our science, but there is evidence aplenty. Dijkstra made pioneering contributions to the design of programming calculi and to the axiomatic method for reasoning about programs. But he seemed unwilling to accept that working to improve programming technology beyond the imperative languages where he made his mark was a worthwhile activity, not only for himself, but for anyone else! His dismissive review of John Backus' Turning Award lecture [15] is a case in point; those interested in exploring this particular issue further should read the archive of the subsequent correspondence between Backus and Dijkstra [11]. Dijkstra's point of view seemed to be that if only everyone were smarter, or thought more, or had more mathematical training, then the deficiencies of our science could be overcome. New technology was not required, and would not help: what was required was a new generation of better trained practitioners.

Dijkstra's is also famous for "writing for himself", if possible by hand with a fountain pen, and eschewing the normal channels of publication in favour of privately distributing his manuscripts, known the world-over as "EWDs".

Van Wijngaarden was from a different mould. He enthusiastically adopted new technology where it would solve a recognized problem, and was ready to pioneer new technology. He was troubled by the inadequacy of BNF (developed for the definition of Algol 60) to express context conditions, and for the definition of Algol 68, he developed a new technology, the two-level grammar, that overcame this deficiency. (These grammars are now known as van Wijngaarden grammars, and have the power of Chomsky type 0 grammars (and thus of a Turning machine), although with a lot more convenience in use. [33]). Van Wijngaarden grammars may not have been the best solution, but they did implement the current practice adopted by static-semantics systems of storing environments of declared variables as concatenated lists, passing type information from the point of declaration to the point of use. Indeed, purely syntactic approaches to type soundness have essentially displaced all others [40, 23]. Two-level grammars also provided a mechanism for uniformly generating productions for sequences of entities, parenthesized entities, and so on, without inventing an unnecessary diversity of special-purpose notations [39]. Our point is that faced with a need, van Wijngaarden was willing to use, or invent technology to address it.

Another instance of this, particularly appropriate in the face of Dijkstra's preference for writing with a fountain pen, is Van Wijngaarden's embrace of the IBM Selectric typewriter. In "A History of Algol 68", Charles Lindsey writes:

> The use of a distinctive font to distinguish the syntax (in addition to italic for program fragments) commenced with [MR 93], using an IBM golf-ball typewriter with much interchanging of golf balls. Each time van Wijngaarden acquired a new golf ball, he would find some place in the Report where it could be used (spot the APL golf ball in the representations chapter). In fact, he did much of this typing himself (including the whole of [MR 101]).

Van Wijngaarden may have done the typing himself because of the inability of the typists at the Mathematisch Centrum to distinguish a roman period "." from an italic period "*.*" [10]).

Eelco was a man very much in the van Wijngaarden mould — in attitudes and interactions, if not in as nattily dressed. He was willing and able to harness technology to get things done. He genuinely cared for those around him, be they students or colleagues. And he created an institution — his research group at Delft — that reified those values.

## 5 Conclusion

In conclusion, it is appropriate to point out that Eelco was a kind man and a sympathetic colleague. His accomplishments may have give him some reason to be arrogant, but to our recollection, he never was. Instead of belittling those who did not or could not follow, he gave them a helping hand. One of us treasures happy memories of a visit to Delft, arranged by Eelco as a means to pay for a trip to SPLASH in Amsterdam, rich with interactions with the members of his group, after which we rode our bikes around Delft with some students. He was endlessly patient as the Grace authors tried to come to grips with Spoofax, and contributed in many other ways to the success of our profession, in particular by supporting SIGPLAN conferences with the conf.researchr.org website, and of course by helping to create and chair IFIP WG 2.16.

Eelco will be sorely missed.

## References

1 Eli Barzilay. Racket, June 2010. `https://blog.racket-lang.org/2010/06/racket.html`.

2 Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

3 Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*, pages 85–98, New York, NY, 2012. ACM. URL: `http://doi.acm.org/10.1145/2384592.2384601`.

4 Andrew P. Black, Kim B. Bruce, and James Noble. Panel: designing the next educational programming language. In *SPLASH/OOPSLA Companion*, 2010.

5 Gilad Bracha. On the interaction of method lookup and scope with inheritance and nesting. In *3rd ECOOP Workshop on Dynamic Languages and Applications*, 2007.

6 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17: A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52–70, June 2008.

**7** Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

**8** Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *Proceedings 44th SIGCSE Technical Symposium on Computer Science Education*, pages 129–134. ACM, 2013. `doi:http://dx.doi.org/10.1145/2445196.2445240`.

**9** Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*, 2013.

**10** Centrum Wiskunde & Informatica. Memories of Aad van Wijngaarden (1916-1987), November 2016. `https://www.youtube.com/watch?v=okLiv1QA4Dg`.

**11** Jiahao Chen. "This guy's arrogance takes your breath away": Letters between John W Backus and Edsger W Dijkstra, 1979. Blog entry, May 2016. `https://medium.com/@acidflask/this-guys-arrogance-takes-your-breath-away-5b903624ca5f`. URL: `https://medium.com/@acidflask/this-guys-arrogance-takes-your-breath-away-5b903624ca5f` [cited 20 Nov 2022].

**12** Luís Eduardo de Souza Amorilm and Eelco Visser. Multi-purpose syntax definition with sdf3. In *Software Engineering and Formal Methods*, 2020.

**13** Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages. In *SLE*, 2018.

**14** Charles Dierbach. Python as a first programming language. *J. Comput. Sci. Coll.*, 29(6):153–154, jun 2014.

**15** E.W. Dijkstra. A review of the 1977 Turing award lecture by John Backus (EWD692). Edsger W. Dijkstra Archive at Univ. Texas, Undated, around November 1978. `https://www.cs.utexas.edu/users/EWD/ewd06xx/EWD692.PDF`. URL: `https://www.cs.utexas.edu/users/EWD/ewd06xx/EWD692.PDF`.

**16** E.W. Dijkstra. Trip report E.W.Dijkstra, Mission Viejo, Santa Cruz, Austin, 29 July – 8 September 1979 (EWD714). Edsger W. Dijkstra Archive at Univ. Texas, September 1979. `https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD714.PDF`. URL: `https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD714.PDF`.

**17** Diwaker Gupta. What is a good first programming language? *ACM Crossroads*, 10(4):7, aug 2004.

**18** Michiel Haisma. Grace in Spoofax. Master's thesis, TUDelft, May 2017.

**19** Michiel Haisma, Vlad Vergu, and Eelco Visser. Grace in spoofax: Readable specification and implementation in one. In *GRACE workshop at ECOOP*, July 2016. `2016.ecoop.org/details/GRACE-2016/2/Grace-in-Spoofax-Readable-Specification-and-Implementation-in-One`.

**20** Michiel Haisma, Vlad Vergu, and Eelco Visser. Spoofax-based implementation of the Grace language, February 2017. `github.com/MetaBorgCube/metaborg-grace`.

**21** C.A.R. Hoare. Hints on programming language design. Technical Report AIM-224, Stanford Artificial Intelligence Laboratory, December 1973.

**22** Michael Homer, Timothy Jones, James Noble, Kim B Bruce, and Andrew P Black. Graceful dialects. In Richard Jones, editor, *ECOOP*, volume 8586 of *LNCS*, pages 131–156. Springer, 2014.

**23** Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001. URL: `http://doi.acm.org/10.1145/503502.503505`.

**24** Laserfiche contributor. How your first programming language warps your brain, n.d. `www.laserfiche.com/ecmblog/programming-languages-change-brain/`, accessed 17 Nov 2022.

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

**26** James Noble and Robert Biddle. programmingLanguage as Language;, June 2021. `https://hopl4.sigplan.org/details/hopl-4-papers/21/programmingLanguage-as-Language-`.

**27** James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. Grace's inheritance. *Journal of Object Technology*, 16(2):2:1–35, April 2017.

**28** James Noble, Michael Homer, Kim B. Bruce, and Andrew P. Black. Designing Grace: Can an introductory programming language support the teaching of software engineering. In *IEEE Conference on Software Engineering Education and Training (CSEE&T)*, 2013. URL: `http://gracelang.org/documents/cseet13main-id92-p-16403-preprint.pdf`.

**29** Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *ESOP*, pages 205–231, 2015.

**30** Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. Language choice in introductory programming courses at Australasian and UK universities. In *SIGCSE*, page 852–857, 2018.

**31** Tiobe index for june 2022. `https://www.tiobe.com/tiobe-index/`, 2022.

**32** Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, 2016.

**33** Adriaan van Wijngaarden. The generative power of two-level grammars. In *ICALP*, 1974.

**34** Vlad Vergu, Michiel Haisma, and Eelco Visser. The semantics of name resolution in Grace. In *DLS*, pages 63–74, 2017.

**35** Vlad A. Vergu, Pierre Néron, and Eelco Visser. Dynsem: A DSL for dynamic semantics specification. In *26th Int. Conf. Rewriting Techniques and Applications (RTA '15*, pages 365–378, 2015.

**36** Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Progr. Research Group, University of Amsterdam, July 1997.

**37** Richard L. Wexelblat. The consequences of one's first programming language. In *SIGSMALL*, page 52–55, 1980.

**38** Wikipedia. Java version history, November 2022. `en.wikipedia.org/wiki/Java_version_history`.

**39** Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *CACM*, 20(11):822–823, 1977.

**40** A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, nov 1994.