The Importance of Being Eelco

Portland State University, OR, USA

Pomona College, Claremont, CA, USA

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, and
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, supported that work withio Spoofax, and provided a venue for discussion within the WG2.16 Programming Language Design working group.

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, Spoofax, syntax

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.4

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, Amazon Web Services, and Agoric.

1 Introduction

The year 2010 seems as if it 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 up to only his second wife, and the idea of Donald Trump as a political figure was far from the mind of the most fevered cartoonist [42].

2010 will also be remembered for the year that OOPSLA was held in a casino in Reno, Nevada. Location aside, the meeting was an important one for the *dramatis personae* of this narrative. The authors of the present paper organized a panel session on the next educational programming language [3], which marked the beginning of our work on Grace. Lennart Kats and Eelco Visser received the Best Student Paper award for a summative article describing Spoofax [30] – hardly the beginning of that work, but a significant landmark; that paper was to receive an OOPSLA Most Influential Paper Award in 2020. Finally, in a restaurant somewhere under a freeway, the first organizational meeting took place of what would eventually become the IFIP Working Group on Language Design, which included Eelco, Andrew, James and (probably!) Kim [21]. Out of that organizational meeting grew plans for the first technical meeting, hosted by Jan-Willem Maessen and Mark Miller in Mountain View the following June.

© Andrew P. Black, Kim B. Bruce, and James Noble;
licensed under Creative Commons License CC-BY 4.0
Eelco Visser Commemorative Symposium (EVCS 2023).
Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 4; pp. 4:1–4:15
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we trace the way that the threads of Spoofax, Grace, and the Working Group on Language Design became woven together over the following twelve years, until Eelco's untimely death in April of 2022. We also explore some of what we believe are core values for language design and implementation: that syntax matters, that programs exist for communication and to be executed, and that the way we treat and are treated by our colleagues makes all the difference.

1.1 What is Spoofax?

The name Spoofax is the result of an engineering exercise conducted by Karl Trygve Kalleberg: "Spoofax" had zero hits in Google, and a free domain name! So wrote Eelco himself in a blog post about the history of Spoofax [49]. Spoofax was a response to the rise of the IDE, which extended the task of a language implementor from simply providing a compiler or interpreter, to also providing a range of editor services, such as syntax highlighting, code completion, outline views, and code formatting. In theory, IDEs such as Eclipse were extensible: all one needed to do was to write the right plugin. In practice, writing a plugin was far from simple. Spoofax *generated* the plugins, starting from declarative specifications of the syntax and semantics of the target language. Spoofax integrated a variety of tools: SDF2 [47] for specifying grammars, generators of customizable editor services based on the syntax, and (initially) the Stratego program transformation language [8] to describe semantics using rewrite rules. Spoofax evolved to encompass additional tools, in particular the DynSem [46] 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.

It's not our goal here to describe Spoofax in detail. Those interested in learning more should start with the above-mentioned blog post [49], and follow up with the double-award-winning paper from 2010 [30].

1.2 What is Grace?

Grace is a programming language intended for the teaching of programming to students beginning the study of computer science. Grace is named after Rear Admiral Grace Murray Hopper, and in the hope that programs written therein would be *Graceful*.

In 2010, the programming languages landscape appeared moribund – especially for academics who were teaching introductory programming courses. According to the TIOBE index [41], Java, C, and C++ were the three most popular languages between 2007 and 2017. Python was slowly bubbling up (from number 8 in 2007 to number 5 in 2017), and the PLT Scheme project was just starting the process of changing the name of their language to Racket [1].

As teachers and academic researchers oriented towards objects, we had to choose a language for our teaching and research [17, 22, 31, 39, 50]. The general dissatisfaction with the options available for teaching came to a head at ECOOP 2010 in Maribor: since we were language designers and implementors, why not work on an object-oriented language targeted at our own needs? Haskell and Algol had been built by teams of academics; why couldn't we design and implement a new object-oriented language for teaching and research? After an initial flurry of interest sparked by the panel-discussion at SPLASH 2010 [38], the task of designing Grace was taken on by Black, Bruce and Noble, the authors of the present paper. The resulting language, as it stood around the time relevant to this article, is described in the short papers presented at SIGSCE [9] and IEEE CSEE&T in 2013 [37].

1.3 What is WG2.16?

Design of all kinds requires feedback. Architecture students learn this early on: they propose and defend their design ideas in charrettes. But computer scientists don't have that tradition: instead we have "publication venues", where we are expected to present our designs as if they were finished, rather than as sketches to invite feedback.

In July 2010, Jonathan Edwards and Andrew Black were discussing this problem in Portland. We roped in Gilad Bracha, and drafted a manifesto:

> A few of us feel that there is currently no good forum for discussing programming language design issues; researchers interested in language design are isolated and lack a place to exchange ideas and criticism. Computer science conferences no longer serve this role, because they have become fixated on rigorous evaluation. There is nothing wrong with rigorous evaluation of things that can be so evaluated, but language design ideas, particularly in their formative stages where feedback is most crucial – and when there may be neither implementation nor experience with their use – cannot be so evaluated. Indeed, many landmark papers on language design could not be published today: Liskov on data abstraction, or Parnas on modules, or Dijkstra on goto. These papers consisted of reasoned arguments for the power of a new idea. We need a venue where such ideas and arguments are still welcome and where they can be refined by constructive criticism.

A number of our colleagues signed on to this manifesto, which eventually because a proposal to IFIP Technical Committee 2 (Software: Theory and Practice) for the creation of a new working group on programming language design. At the zeroth meeting in Mountain View in 2011, Eelco Visser was elected as chair of this loose group. Eeclo took the proposal to TC2, gained their approval, and the group become IFIP WG2.16 – the Working group on Programming Language Design

2 A Little Grace

Our interactions with Eelco were primarily in the context of the Grace programming language design project. While paper is not a primer on Grace, some discussion of two of Grace's peculiarities is necessary to provide background for the story of our interactions with Eelco and the Spoofax team.

2.1 Objects and Classes

A Grace object is created by an object constructor, a special kind of expression introduced by the reserved word **object**. A Grace **class** is a declaration that names and parameterises an object constructor. Here is an example:

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

```
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."
}
dog "Fido"
```

The code on the left is an expression that creates a new object. On the right is a class declaration; it declares a *method* named dog(_) whose body is a parameterized version of the object constructor on the left. The request dog "Fido" requests execution of this method, and thus creates a new object. Both fragments print "Fido has been born."

2.2 Syntax and Layout

Because Grace was intended for teaching, its syntax was designed to be writable by novices and readable by anyone familiar with another object-based language. We chose a relatively conventional mix of the "curly bracket" style of C and the keyword style of Pascal. Like C, declarations and code blocks are delimited by {...} rather than begin...end, but like Pascal, all 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. Newlines terminate statements, making semicolons optional, and eliminating a common source of frustration. Control structures are not built in; instead they are methods that use Grace's multi-part method names, inspired by method names in Smalltalk but modified to eliminate the colons. As a consequence, keywords appear between the components: "if (flag) then { ...}" rather than "if (flag) { ...}".

We debated long and hard whether Grace should be a "curly bracket language" like C or an "indentation-sensitive" language like Python. We ended up requiring both! We chose braces to delimit blocks, because Grace blocks are λ-expressions, and we wanted a compact in-line form for simple function such as $\lambda x.x + 1$, which is written in Grace as {x → x + 1}. The story in Grace is that "braces are suspenders": code enclosed by braces is not executed immediately, but is stored away so that it can be executed later (i.e., is represented as a closure). This is consistent with the use of braces to enclose method bodies.

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. Here is a short example:

```
1  def direction = if (a > b) then {
2      "up"
3  } elseif { a < b } then {
4      "down"
5  } else {
6      "fixed"
7  }
8  while {stream.hasNext} do {
9       print(stream.read)
10 }
```

The expression on the right-hand side of the **def** on line 1 uses parentheses to enclose the condition (a > b), because that condition is always evaluated, and indeed is evaluated before the if(_)then(_)elseif(_)then(_) is executed. However, the blocks after then, elseif, and else must be evaluated only when necessary: it is to make this possible that they are wrapped in braces. The while(_)do(_) statement is similar: because it requires repeated evaluation of both the while condition and the body of the do, they must both be wrapped in braces.

While indentation must correspond to the brace structure, it is also used to determine the span of statements, and thus the elision of semicolons. In particular, indentation distinguishes between a single method request with a multi-part name, and multiple requests with single

part names. The left and centre layouts below will both work for Grace's if(_)then(_)else(_) control structure: the one on the left indents the blocks following then and else, while the center example used indentation *not* following an open brace to turn the three physical lines into a single logical line. In contrast, the rightmost layout will be interpreted as three separate method requests, on three separate lines: an if(_), a then(_), and an else(_):

```
if (condition) then {
    doThis
} else {
    doThat
}
```

```
if (condition)
    then { doThis }
    else { doThat }
    //one request
```

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

2.3 Nesting and Inheritance

Grace objects and classes can inherit from other classes via **inherit** (and the closely related **use**) clauses. For example: we can define an object out that inherits from a superClass:

```
class superClass {
    method m { "in superclass." }
}
def out = object {
    inherit superClass
    method foo { print (m) }
}
out.foo
```

The program above will print "in superclass." Note that Grace follows C++, in allowing the programmer to elide **self** in method requests, and Eiffel and Self with a single syntax and namespace for both methods and fields. So a request of m (meaning **self**.m) in the body of method foo is an example: here that resolves to the foo method declaration in superClass: but the same request could equally resolve to a constant definition, to reading a variable, or even instantiating a class.

In addition, Grace's objects – like those in most contemporary object-oriented languages following BETA [32] – can be lexically nested:

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

The program above will print "in enclosing object." Note the method foo is lexically 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 as an earlier draft of this paper said instead: "the ogre is in the orthogonality". Although comically alliterative, that remark is inaccurate: it is the sort of remark that gives orthogonality an unjustifiably bad reputation (see Section 4.3). A moment's thought should make us realize that nesting and implicit **self** **cannot** be orthogonal: implicit **self** is dubious [34] in a context in which there are multiple current objects.

What if a program attempts to use nesting, inheritance and implicit **self** at the same time, as in the example below?

```
1  class superClass {
2      method m { "in superclass." }
3  }
4  def out = object {
5      method m { "in enclosing object." }
6
7      def inner is public = object {
8          inherit superClass
9          method foo { print (m) }
10     }
11 }
12 out.inner.foo
```

Which m does the method foo invoke: the m in the lexically-enclosing object out, or the m inherited from superclass? This is exactly the sort of question that a language workbench such as Spoofax should help one explore.

This potential ambiguity is common across many object-oriented languages – with as many different solutions as there are languages [7]. 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 bans such ambiguous requests, requiring that the programmer resolve the ambiguity by writing **self**.m or **outer**.m, as shown in Figure 1 [6, §implicit-requests]. Nonetheless, the fact that the question "Which m?" can be asked at all means that a complete specification of the language must answer it.

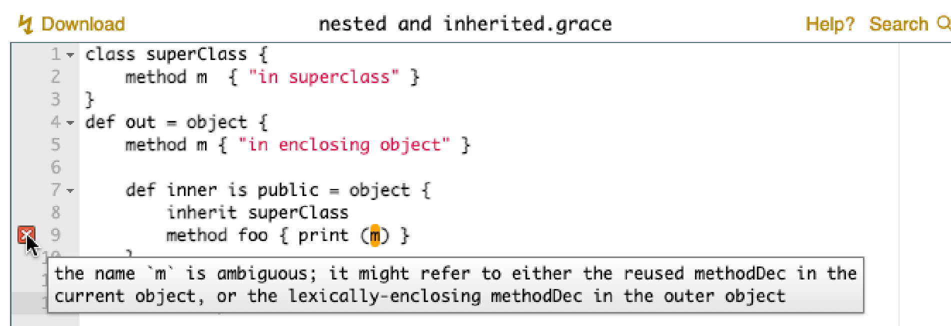

Figure 1 Grace IDE showing ambiguous implicit request.

3 Grace in Spoofax

The original goal of the Grace project was to produce a language specification, not a language implementation [4, 5, 10]. While at least one implementation would be essential to guide the process of writing the specification, we hewed to the 20th century ideal that a programming language should be implementation independent. Build it (the specification), we thought, and they (the implementors) will come. How naïve we were![1]

[1] Not just naïve, but historically wrong: pretty much every successful programming language since SIMULA has been based on a single canonical implementation.

3.1 SDF2 Grace Parser

But come they did, or rather, Eelco Visser and his Spoofax team came: notably master's student Michiel Haisma 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" [5], but as one of the forces behind what became IFIP WG2.16, he was part of the audience for 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 between Eelco and James Noble, following up on a "conversation after the pub", Eelco expressed interest in becoming an early implementor of Grace. We were of course delighted: as Eelco's involvement promised incisive design feedback and a soild implementation – while Eelco would benefit from practical experience with Spoofax on a language not of his own design. Indeed, Eelco went as 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 risks associated with that, so don't hold your breath.[2]

Eelco was as good as his word. Working off an early grammar for Grace (at the time, represented using a parser combinator library within Grace itself) by SPLASH in October 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 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.

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 [23, 24, 45].

Figure 2 shows the architecture of Spoofax–Grace. Spoofax's SDF3 DSL [15] parses Grace code into an initial AST. Next, the Stratego transformation language rewrites some Grace constructs (such as classes and traits) that are 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 [8]. Finally, definitions in the DynSem DSL [46] are used to interpret the program represented by the lowered AST.

[2] Email message from Eelco Visser to James Noble dated 4 March 2012.

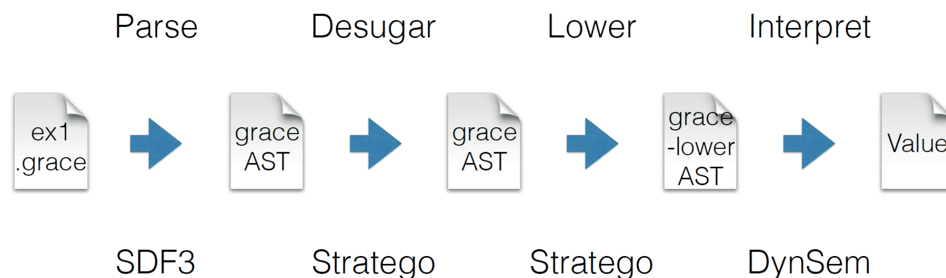

Figure 2 Spoofax–Grace Architecture. From Michiel Haisma, "Grace in Spoofax" [23], used with permission.

Figure 3 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 name resolution: the part of the language that had to combine the local definitions within an object with the definitions inherited from superclasses, reused from traits, and located in enclosing lexical scopes, which include the the outermost scope that defines the module's dialect [27, 36]. Based upon Eelco's theory of Scope Graphs [35, 43], Grace's name resolution and request lookup semantics were encoded as part of the overall operational semantics using the Spoofax DynSem DSL [46]. The DynSem implementation was shorter and easier to modify than the existing Grace implementations [45], 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 $O(np)$[3].

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 type system, it was important throughout our design discussions 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. However, the early specification documents didn't make this sufficiently clear, and Vergu *et al.* write: "The use of expressions to determine ancestors means that meaningful name resolution can only be performed at run time" [45]. 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 clear that we needed to do some serious work on our language specification – in particular, to reconsider the definition of *manifest*.

[3] To see why, consider an object n levels deep, where each enclosing defintion inherits from p classes or traits. This means there are $O(np)$ possible positions where m could potentially be declared – basically once in every lexically enclosing definition, and once in every definition inherited into each lexical scope. Grace's rules about ambiguity and (lack of) overloading means that every potential position must be checked, not just the positions that are inhabited.

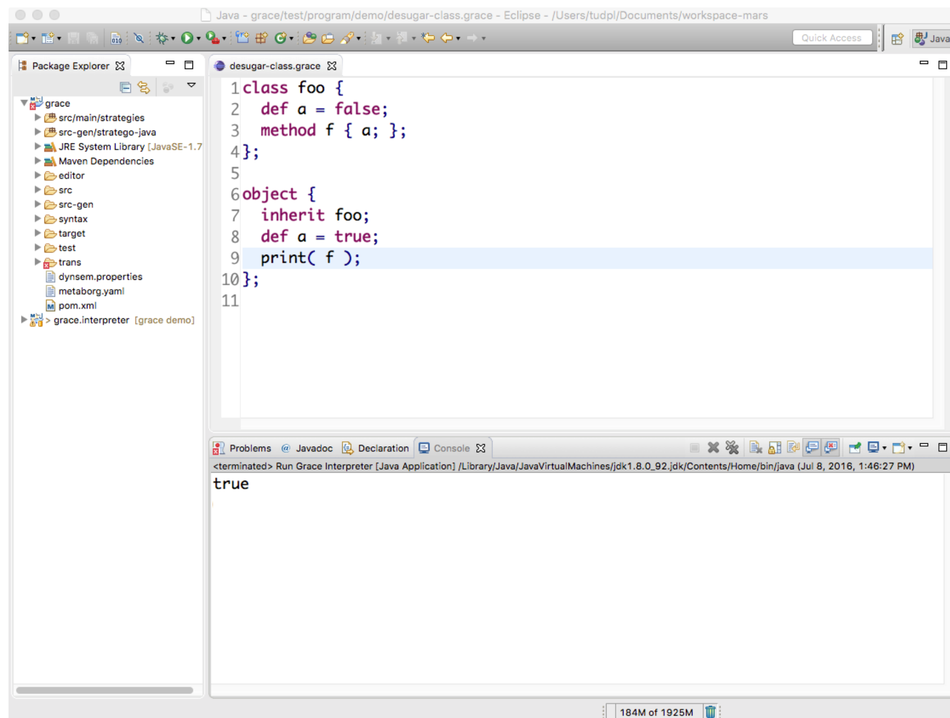

Figure 3 Grace running in Eclipse, using plugins generated by Spoofax. From Michiel Haisma “Grace in Spoofax” [23], used with permission.

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. The Spoofax implementation of Grace is still available [25], although not maintained, and now includes a parser that can handle Grace's layout, based on extensions to SDF3 that were made while the main Spoofax–Grace project was coming to an end [15, 16].

Looking back over this collaboration, it's clear the result was successful for both parties. Eelco's team obtained valuable practical experience, which eventually led to extensions to SDF for layout parsing; while Grace gained significant feedback on both the langauge design and its presentation in the language specification document. One important lesson was that a clear separation between static and dynamic semantics would 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, when intended, this should be done explicitly, e.g.: “The checks necessary to implement this guarantee [type safety] may be performed statically or dynamically”. Another lesson – in a quite different dimension – is the value of the Agile practice of the on-site customer [2, p. 60]: if the Grace and Spoofax teams had been co-located, the lack of clarity about what could be inherited would have been discovered much sooner.

4 Being Eelco

This paper makes some presumptuous claims about the core values of language design and implementation, and the way in which Eelco Visser embodied those values. If Eelco were still with us, we could do this 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 the reader to decide if they have value.

4.1 Semantics vs. Syntax

We are going to say it outright: syntax is important! We do not claim that semantics are *unimportant*, but that syntax must come first: the semantics has to be attached to something. Syntax *carries* the semantics: the vast majority of programmers will only approach semantics via syntax [12, 14]. A well-designed syntax should suggest the appropriate semantics; if a naïve reading of the syntax is at variance with the semantics, then one or the other needs to change.

In Spoofax, Eelco acknowledged that syntax matters to programmers. Parsing, pretty-printing, and editor support are important. 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 set aside a lot of what legitimately concerns users, and abdicates 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" [20]. Having an extensive library was apparently a fatal flaw in Dijkstra's eye[4].

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, 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: Spoofax 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 it will still parse all of one's test cases. 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

[4] "For the absence of a bibliography I offer neither explanation nor apology."
Preface to Dijkstra's *A Discipline of Programming* [18].

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.

We wish that we could write: "And such an implementation is just what Eelco and Spoofax had provided." Unfortunately, the timing did not work out. Because the Spoofax implementation did not originally recognize Grace's indentation rules, and all of our existing code used indentation rather than semicolons, we could not run our code on Spoofax-Grace. Our design was indeed guided by an implementation, but not one created in Spoofax. Instead we used a self-hosting compiler written in Grace itself, based on an early implementation written by Michael Homer, who at the time was a student at Victoria University of Wellington [26]. A self-hosting compiler had some advantages – it gave us experience with a substantial body of real Grace code – but did not provide a readable syntax or semantics. Nevertheless, we derived great benefit from the Spoofax implementation. Designing a language can be a lonely business; the fact that another team in another country were *interested* in what we were doing, to the point of taking our specification and implementing it, was heartening and rewarding.

4.3 Dijkstra vs. Van Wijngaarden

Although Adriaan van Wijngaarden was Edsger Dijkstra's boss and doctoral supervisor, the two men could hardly have been more different. Let us concentrate here on just one difference: where Van Wijngaarden was an enthusiastic adopter of new technology, Dijkstra seems to have been less interested in what he called "gadget development"[20].

This may appear to be an odd comment to make about one of the pioneers of our science, but there is evidence aplenty. Dijkstra made original contributions to the design of programming calculi and to the axiomatic method for reasoning about programs. Dijkstra, however, 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 [19] 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 [13]. 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.

Van Wijngaarden was from a different mould. He aimed to contribute to a group, rather than work as as solitary individual. He enthusiastically adopted new technology where it would solve a recognized problem, and was ready to try out new ideas. He was troubled by the inability of BNF (developed for the definition of Algol 60) to express context conditions; for the definition of Algol 68 he developed a new technology, the two-level grammar, that overcame this deficiency [44].

In the years since their invention, two-level grammars, also known as W-grammars after Van Wijngaarden, have acquired a bad reputation. The notion of "orthogonality" in language design, much facilitated by the use of a W-grammar for language description, has faired little better. Perhaps this is because the adoption of W-grammars by WG2.1 for the description of what became Algol 68 can be seen as the beginning of the schism that led to the Algol 68 minority report and significant resignations from WG2.1. Nevertheless, it's worthwhile to look at the *ideals* carried by W-grammars and Van Wijngaarden's notes on "*Orthogonal design*" (MR76) [44]:

1. That all the data values that can be manipulated in the programming language should be "first class". In other words, if you can put one type of value (such as a number) into a list, or pass it as an argument, or return it as a result, then you should be able to do the same with other types of values (such as strings, records, and functions).
2. That the effect of declarations should be recorded in a syntactic structure, along with their types, and later references to declared names should be looked up in this structure to ascertain whether or not they have been declared, and if so, to find their types.
3. That languages can be *expression-oriented*, in that every construct can be considered as an expression whose execution delivers a value of some type (as well as having a possible effect on the store), rather than distinguishing between statements and expressions.

Independent of whether we like them or not, all of these contributions have won in the marketplace of ideas. Indeed, purely syntactic approaches to type-checking [28, 52] (Item 2 above) have essentially displaced all others. Two-level grammars also gave Van Wijngaarden a mechanism for uniformly generating productions for *sequences* of entities, *parenthesized* entities, and so on, without inventing an unnecessary diversity of special-purpose notations [51]. Our point is that faced with a need, Aad van Wijngaarden was willing to use, or invent, technology to address it.

What then is the problem with W-grammars? Primarily, that they are *too* powerful. As early as 1967, Michael Sintzoff [40] showed[5], that any recursively-enumerable set can be described by a W-grammar. This means that the parsing problem for W-grammars is *in general* undecidable. Of course, when using a W-grammar to describe a programming language, the author *intends* that the language be parsable, and there have been many attempts to place formal restrictions on W-grammars to ensure that parsing is possible. Early in his career (1998), Eelco authored one such attempt [48], which also makes useful connections between W-grammars and algebras, and between orthogonality and polymorphism.

Another instance of Van Wijngaarden's willingness to embrace technology, particularly appropriate as a contrast to Dijkstra's preference for writing with a fountain pen, is Van Wijngaarden's use of the IBM Selectric typewriter. In "A History of Algol 68", Charles Lindsey writes [33]:

> 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 "*.*" [11]. The point is that Van Wijngaarden was willing to do relatively mundane work himself, adopting the latest technologuy to do so.

Eelco Visser was a man very much in the Van Wijngaarden mould – in attitudes and interactions, if not as nattily dressed. (As far as we can ascertain, they are not related academically, other than both being Dutch.) He was willing and able to harness technology to get things done: Eclipse is, after all, merely the 2010s version of a golf-ball typewriter [29]. He created institutions – his research group at Delft WG2.16, – that reified those values. He

[5] We should write: "is reputed to have shown" because we haven't actually found a copy. However, Eelco references this work in his 1998 paper [48], so perhaps that's good enough?

built tools, not just Spoofax, but also WebDSL, and the researchr conference system, that meant other people could get their own work done. He genuinely cared for those around him, be they students or colleagues.

5 Conclusion

Eelco Visser was a kind man and a sympathetic colleague. His many accomplishments never went to his head. Instead of belittling those who did not or could not follow, he gave them a helping hand. Eelco was endlessly patient as the Grace authors tried to come to grips with Spoofax.

Andrew Black treasures happy memories of a visit to Delft, arranged by Eelco as a means to pay for a trip to SPLASH in Amsterdam. The visit was rich with interactions with the members of his group, after which we and some students rode our bikes around Delft *en route* to dinner. Eelco 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 WG2.16. Eelco is sorely missed.

References

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

2 Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition, 1999.

3 Andrew Black, Kim B. Bruce, and James Noble. Designing the next educational programming language. In *Proceedings ACM international conference on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 201–204, New York, NY, USA, October 2010. ACM. `doi:10.1145/1869542.1869574`.

4 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. `doi:10.1145/2384592.2384601`.

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

6 Andrew P. Black, Kim B. Bruce, and James Noble. Grace language specification, version 0.8.2. `http://web.cecs.pdx.edu/~grace/doc/lang-spec/`, July 2020.

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

8 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, June 2008.

9 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:10.1145/2445196.2445240`.

10 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.

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

12 Daniel Chandler. *Semiotics: The Basics*. Routledge, 2002.

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

14 Paul Cobley and Litza Jansz. *Semiotics for Beginners*. Icon Books, Cambridge, England, 1997.

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

16 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.

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

18 E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

19 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. URL: https://www.cs.utexas.edu/users/EWD/ewd06xx/EWD692.PDF.

20 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. URL: https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD714.PDF.

21 Jonathan Edwards. Reno 2010 [online]. October 2010. URL: https://languagedesign.org/meetings/Reno2010.html [cited Jan 2023].

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

23 Michiel Haisma. Grace in Spoofax. Master’s thesis, TUDelft, May 2017.

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

25 Michiel Haisma, Vlad Vergu, and Eelco Visser. Spoofax-based implementation of the Grace language, February 2017. URL: https://github.com/MetaBorgCube/metaborg-grace.

26 Michael Homer. *Graceful Language Features and Interfaces*. PhD thesis, Victoria University of Wellington, 2014.

27 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.

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

29 Poul-Henning Kamp. Sir, please step away from the ASR-33! to move forward with programming languages we need to break free from the tyranny of ASCII. *Queue*, 8(10):40–42, October 2010.

30 Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proc. ACM Int. Conf. Object Oriented Programming Systems Languages and Applications*, OOPSLA ’10, pages 444–463, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869497.

31 Laserfiche contributor. How your first programming language warps your brain [online]. Undated. URL: https://www.laserfiche.com/ecmblog/programming-languages-change-brain.

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

33 C. H. Lindsey. A history of Algol 68. In *History of Programming Languages – II*, pages 27–96. Association for Computing Machinery, New York, NY, USA, 1996. doi:10.1145/234286.1057810.

34 Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP Proceedings*, 1999.
35 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *ESOP*, pages 205–231, 2015.
36 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.
37 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`.
38 Jack Rosenberger. Grace: A manifesto for a new educational object-oriented programming language. Blog@CACM, October 2010. Retrieved Jan 2023. `http://cacm.acm.org/blogs/blog-cacm/100389`.
39 Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. Language choice in introductory programming courses at Australasian and UK universities. In *Proc. 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 852–857, 2018.
40 Michel Sintzoff. Existence of a Van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société scientifique de Bruxelles, Série 2*, 81:115–118, 1967.
41 TIOBE Index for June 2022. `https://www.tiobe.com/tiobe-index`, 2022.
42 G.B. Trudeau. *Yuge!: 30 Years of Doonesbury on Trump*. Andrews McMeel, 2016.
43 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.
44 A. van Wijngaarden. *Orthogonal design and description of a formal language*. Rekenafdeling: MR. Stichting Mathematisch Centrum, 1965. MR 76. URL: `https://ir.cwi.nl/pub/9208/9208D.pdf`.
45 Vlad Vergu, Michiel Haisma, and Eelco Visser. The semantics of name resolution in Grace. In *DLS*, pages 63–74, 2017.
46 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, June–July 2015.
47 Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Progr. Research Group, University of Amsterdam, July 1997. URL: `https://eelcovisser.org/publications/1995/Visser95.pdf`.
48 Eelco Visser. Polymorphic syntax definition. *Theoretical Computer Science*, 199(1–2):57–86, 1998. `doi:10.1016/S0304-3975(97)00268-5`.
49 Eelco Visser. A brief history of the Spoofax language workbench [online]. February 2021. URL: `https://eelcovisser.org/blog/2021/02/08/spoofax-mip/` [cited January 2023].
50 Richard L. Wexelblat. The consequences of one's first programming language. In *SIGSMALL*, pages 52–55, 1980. `doi:10.1145/800088.802823`.
51 Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *CACM*, 20(11):822–823, 1977.
52 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.