The Design of Elegant Languages

Lambert Meertens

Department of Algorithmics and Architecture, CWI, Amsterdam, and
Department of Computing Science, Utrecht University, The Netherlands

0 Introduction

It was a dark and stormy week in Munich, the third week of December 1968, in which IFIP Working Group 2.1 decided to submit MR 100 as "the consolidated outcome of the work of the Group". MR 100 was the document describing the design of the Algorithmic Language ALGOL 68 [4], after many iterations, with Aad van Wijngaarden's MR 76 [3] as the starting point.

The weather outdoors was fair for the time of the year, a crisp cold; but darkness had descended upon the hearts of the Working Group, and storms were raging in the hall of the *Bayerische Akademie der Wissenschaften* where WG 2.1 was assembled. A substantive minority of the members had strong criticism of the outcome of the whole enterprise; so strong in fact, that it was apparently impossible to discuss the alleged shortcomings of the proposed language[0] in a technical way, let alone suggest improvements that might result in a design that could have found grace in their eyes[1].

The criticism was laid down in a Minority Report, signed by Edsger Dijkstra, Fraser Duncan, Jan V. Garwick[2], Tony Hoare, Brian Randell, Gerhard Seegmüller, Wład Turski and Mike Woodger.

The following is quoted from the Minority Report.

> Now the language itself, which should be judged, among other things, as a language in which to *compose* programs. Considered as such, a programming language implies a conception of the programmer's task. [...] More than ever, it will be required from an adequate programming tool that it assists, by structure, the programmer in the most difficult aspects of his job, viz. in the *reliable creation* of sophisticated programs. In this respect we fail

0. There was also strong criticism of the "how" of the description itself next to the "what" of what was described, but I shall leave this aside.

1. In the interest of fairness it should be pointed out, however, that Fraser Duncan had before produced proposals for more important changes, and although these were not followed he remained very active up to the end in reporting technical deficiencies and suggesting minor improvements, most of which were indeed incorporated.

2. Garwick was actually not present at the meeting, but requested afterwards that his name be included among the signatories. Also not present was S.S. Lavrov, who in fact had drafted an early version of the Minority Report.

> to see how the language proposed here is a significant step forward: on the contrary, we feel that its implicit view of the programmer's task is very much the same as, say, ten years ago. This forces upon us the conclusion that, regarded as a programming tool, the language must be regarded as obsolete.

What I set out to do here next, is to look at programming languages from a conception of the programmer's task and deal with some aspects in the evolution of programming languages viewed, specifically, as languages in which to *compose* programs. The treatment reflects largely my personal experience and taste in programming, and as such will not at all be comprehensive. In doing this I shall pay particular attention to ALGOL 68. It is, however, not my aim to give a "critical but balanced" assessment of this language. Also, I will freely ascribed "innovations" to language B, even though it may be argued that the essence of the idea existed before in language A, if B was the first to do it right, or with sufficient generality.

1 Limitations of the human mind

Except for rare cases, programs are not written in machine language, but in some programming language. Nowadays one important aim of using a (commonly available) programming language is to achieve program portability. This is an aspect that I shall not consider here. I want to look at programming languages here purely as languages in which to *compose* programs.

As such, a programming language is undeniably a tool, and, following the Minority Report, we can require of an adequate programming tool that it assists, by structure, the programmer in the most difficult aspects of his job, viz. in the *reliable creation* of sophisticated programs. It is not necessary to give a detailed analysis of the programmer's task to agree that the statement that this is difficult (and difficult it is) is also a statement about limitations of the human mind. If we had no such limitations, we would not need computers or programs to start with.

The main limitations we have that are relevant here are probably the following three.

In the first place, human long-term memory is bad for remembering "meaningless" things, like telephone numbers or nonsense text. If we do not use or at least recall something like that daily, we tend to forget it. For meaningful things that we do recall, we often make substitution errors. For example, we may recall the phrase "Well, he's probably pining for the fjords" as "He's pining for the fjords, you know".

Secondly, the amount of things that we can mentally handle *simultaneously* is severely limited. As an example, try (without writing down intermediate results) substituting simultaneously $a+b$ for each occurrence of a and, likewise, $a-b$ for each occurrence of b in the formula $(2a+b)(a-2b)-(a+b)(a-3b)$ while at the same time expanding the result by "multiplying it out"[3].

Finally, when we have to perform some routine task repeatedly, we lose attention

3. For those who want to try this, here is a simple check on the result. Evaluate the resulting formula for $a=2$ and $b=1$. If the outcome is not 7, something went wrong.

after some time and start making the silliest clerical errors imaginable, precisely in those things we understand and know perfectly well.

In the best case, a programming language helps the programmer to cope with the task of program construction by offering ways to get around such limitations. A simple example is the programmer's ability to choose meaningful identifiers, such as "*vector*", "*velocity*" and "*version*", instead of "*V*", "*V1*" and "*V2*". In a program with hundreds and hundreds of identifiers — not at all uncommon — this is of inestimable help. In the worst case, programming is a struggle with the programming language itself, a struggle in which more mental effort is spent in trying to cope with the intricacies and idiosyncrasies, if not idiocies, of the language, than on the actual problem.

2 The programmer's task

Part of the difficulty of programming, and sometimes the most difficult part, may be to decide what the program is to do in the first place, rather than how this is to be done. For example, in creating a good code generator, the hard task is to decide and specify what code it will generate; after that the actual "coding", although perhaps not entirely trivial, is definitely only a small part of the whole problem.

Inasmuch as I have witnessed grandiose failures of software projects, fortunately usually from a comfortable distance, these were always foreshadowed by a failure to get a clear position on the "what", or even to reach agreement between the actors involved as to the basic objectives of the project.

Various kinds of formalisms can help to record the decisions taken, if any, but as far as I am aware they tend not to be particularly useful in reaching these decisions, and I see no clear role here for what I consider to be programming languages.

Deciding on the "what" cannot be entirely separated from the "how": sometimes a small change in the specification that is almost irrelevant from the point of view of the user of the program may be the difference between entirely feasible and entirely infeasible. For example, in code optimization, there are fast and reasonably simple techniques that give very good register allocation, but for obtaining a truly optimal allocation — only marginally better than very good — the fastest algorithms we have may easily take more time than one could reasonably hope to gain by the optimization.

Let us, however, assume that the "what" is given. The next step is to go from the "what" to the "how", which typically involves designing, jointly, suitable data structures, and algorithms operating on data encoded in terms of such structures[4]. In my experience, this part is only rarely a difficult job. Typically I see "immediately" some obvious approach. Usually there is an "obvious" decomposition into sub-problems for which rather standard data structures and straightforward algorithmic techniques will do the job. (It helps, of course, to have some knowledge of and experience with such techniques.) In the rare cases in which the approach is not obvious, there is at least the satisfaction of an intellectual challenge.

So where we are now is that in principle the algorithms to be used, including the relevant data structures, are sufficiently clear in the programmer's mind, and that the task

4. I am not considering the design of distributed programs, which requires a very different approach.

at hand is to cast these abstract but clear ideas into the form of a program; to create a concrete embodiment of the algorithms and data structures by means of a specific programming language. It is — still in my experience — here that suffering starts, and that that which was so clear becomes obscure, if not a mess.

As a first step a mapping of the abstract data structures has to be given in terms of the available data types of the programming language. The mapping has to be such that the basic operations in terms of which the abstract algorithm is formulated can be implemented efficiently. This may involve explicit allocation and deallocation, requiring extensive administration to keep track of what is being used when by whom.

The implementation of the basic operations can be viewed as programming tasks on their own, creating as it were an abstract machine on top of which the abstract algorithm is implemented. Ideally, the effect is that the programming language is extended with new data types, in a way as if the language had been designed with these data types built-in from the start. In the point of view in which a program is seen as emerging from its proof, this separation corresponds to a factorization of the proof by separation of concerns. Whereas this was a bottom-up phase, expressing the abstract algorithm in these primitives is largely a top-down process, corresponding to the (possibly repeated) decomposition of the problem into sub-problems.

Now where do the problems come in? What is it that makes this so difficult? Let me sketch the worst-case scenario. Assume that the language is such that no suitable interface can be created between the "abstract machine" and the implementation of the abstract program because its abstraction and encapsulation mechanisms are too deficient. This means that the concrete implementation of the abstract operations has to be spelled out again and again. The inbuilt data types are weak, so that this implementation is complex and involves awkward bookkeeping, which has to be woven through the program under construction. So all the time the programmer has to keep two different abstraction levels in mind, each with their own meaning, representations, and invariants. Moreover, the programming language offers little textual support for composing programs from sub-programs solving sub-problems, so that the programmer also has to keep track of where the decomposition is in the traversal of a virtual tree, and this possibly again simultaneously at different abstraction levels. This is by no means the end of the suffering of our poor programmer. By the time this process reaches the point where a construct from the programming language can actually be used (pant pant), the question arises what its concrete syntax is. What was its name again? Was this keyword abbreviated or not? Are the separators here commas or semicolons? What was the order of the parameters? Then, are there perhaps restrictions that apply here? Or some semantic exception?

With this scenario the programmer has to keep, all the time, a large amount of detail in mind, while continually retrieving confusing items from long-term memory, in order to perform a rather mindless task, the repeated expansion of substituting the concrete implementation in the abstract operations, and weaving through that the bookkeeping code.

Almost impossible. And yet this is what most programmers do all the time. For most programming languages, and certainly those that have a good deal of currency, suffer to some extent from almost all of these problems.

3 On elegance

What makes some designs more elegant than others? In general, when we call a design elegant, we mean that the design displays ''good taste'', both in the choice of the elements of which the design is composed and in the way they are combined. So elegance is — to a certain extent, but undeniably — a matter of taste. A well-known saying tells us that there is no accounting for taste, and indeed, discussing such an elusive æsthetic judgment as elegance in an academic context is a somewhat precarious enterprise. Yet, as I shall argue, elegance in programming-language design is a difficult but important aim.

The notion of ''elegance'' can be clarified somewhat further by some reflection on what we would, definitely, consider inelegant. Something can be inelegant because of ''too-muchness'': when the design is suffering from an excess of elaboration, with too many frills. Inelegance can also be due to a lack of balance, which can be defined as a local ''too-muchness''. Finally, a design can display bad taste in the incompatibility of its components, for example in style.

An elegant design, then, is one that is characterized by the apparent simplicity with which its effect is obtained, evidenced by a certain restraint in the choice of elements, in number as well as in style: no individual part may give the impression it is superfluous, and the overall design should give the impression of a conceptual unity.

Most programming-language designers will argue — or so I expect — that simplicity is a desirable property of programming languages, and that their own pet language is simple. Now there is simple and simple. A knife is simpler than a pair of scissors. But clipping an article from a newspaper using scissors is a simpler task than cutting it out with a knife. It is meaningless to apply a notion like simplicity to a programming language without reference to its use as a tool for constructing programs, and without considering in particular the nature of that task in relation to the limitations of the human mind.

An essential aspect of that task is that the ''features'' of a language are not *individually* used as tools the way a carpenter uses a plane and then puts it aside to use next a chisel, and so on. It is the very act of combining various elements by which the program is constructed.

From the above discussion of the programmer's task, we can see that all kinds of ''bells and whistles'' do more harm than good, and that, more than many ''ready-made power features'', we as programmers need a careful choice of elements that lend themselves to easy and graceful combination. An important part of that is that the rules for what may be combined when and how are easy. Of paramount importance for composability is the presence of abstraction mechanisms by which interfaces between abstraction levels can be created.

We see thus how the element of elegance comes in. A language in which these things come together nicely will be felt by its users to be elegant. A somewhat different viewpoint is that in which programs are constructed by deriving them formally. However, the difference is more apparent than real. Precisely the same properties are required to make this formal activity doable, and in fact even more so.

4 Before ALGOL 60

The evolution of programming languages, already before ALGOL 60, has been away from the low-level model provided by hardware architectures, creating instead a more elegant abstract machine model. This evolution started with so-called assemblers, which are traditionally viewed as giving a direct mapping to machine code but with some unpleasant chores having been taken over, and with mnemonics to aid the programmers. However, the difference between assembly languages and higher-level languages is not necessarily a matter of principle. For example, in C the underlying hardware with its linear memory model as a contiguously addressable sequence of words keeps staring us in the face. Conversely, assembler languages may have nice abstraction mechanisms that are a valuable help in programming. It is entirely possible than to program as if the language was high-level.

Already there we find that details of the mapping to the concrete machine that are not relevant are taken away from the responsibility of the programmer. A quantity has to be stored somewhere, and the same location should not be used for something else during its lifetime, but which actual location is chosen is completely irrelevant. The higher the language level, the more such irrelevant things tend to get hidden. The main innovations of FORTRAN were an explicit parameter mechanism, automatic mapping of more-dimensional arrays to linear memory, and — whence its name — "formula translation".

5 The contribution of ALGOL 60

Suppose we have to write a program for some task *T*, which is too complex to be expressed directly as a basic step. However, whenever a certain condition *C* is satisfied, the task *T* can be reduced to the simpler task *T0*, whereas otherwise it can be simplified to *T1*. So, assuming we have programs for *T0* and *T1*. we can now give one for *T*. In FORTRAN such as it was when ALGOL 60 was being designed, conditional execution was achieved by using "conditional jumps" in the program. This language construct was modelled rather straightforwardly after the same low-level machine-code instruction on the IBM 704. Using ALGOL 60 syntax style, we get then:

```
        if not C then goto L1;
        T0;
        goto L2;
L1:     T1;
L2:
```

If *T0* and *T1* are still too complex and are likewise decomposed into subtasks, and so on, we arrive at the third step at the following:

```
        if not C then goto L1;
        if not C0 then goto L01;
        if not C00 then goto L001;
        T000;
        goto L2;
L001:   T001;
        goto L2;
L01:    if not C01 then goto L011;
        T010;
        goto L2;
L011:   T011;
        goto L2;
L1:     if not C1 then goto L11;
        if not C10 then goto L101;
        T100;
        goto L2;
L101:   T101;
        goto L2;
L11:    if not C11 then goto L111;
        T110;
        goto L2;
L111:   T111;
L2:
```

The result looks like spaghetti but is rather more unpalatable. It is far from easy to see from this text under what conditions exactly *T101*, for example, will be executed. It is, of course, not accidental that this resembles compiled code. It *is* compiled code: hand-compiled from the abstract idea into to the limited constructs of a programming language.

A major innovation of ALGOL 60 was that specific program-composition constructions were provided for the common cases of task decomposition. For conditional execution ALGOL 60 has

```
if C
then T0
else T1
```

With this notation, the above spaghetti program becomes:

```
if C
then if C0
     then if C00
          then T000
          else T001
     else if C01
          then T010
          else T011
else if C1
     then if C10
          then T100
          else T101
     else if C11
          then T110
          else T111
```

The advantage should be clear.

As a result, the definition of what a permissible program is got a recursive nature. To describe this exactly, the authors of the ALGOL 60 Report had to invent[5] a new grammatical formalism, which became known as BNF. Compiler writers tend, for reasons of their own, to like silly restrictions that make the programmer's life harder, such as: identifiers may have only seven characters; or: procedures may not be nested inside other procedures; or: expressions may only be nested five levels deep and contain at most 511 subexpressions; and so forth. In doing serious programming, programmers keep running into such restrictions, and getting around them may take more than half of the coding effort, if it is possible at all. The total amount of time wasted this way is several orders of magnitude more than the time gained by the lazy compiler writer. If the syntax of a language is described by "verbal prose", such restrictions are easy to put in. Using BNF, it is easier to describe a language without such arbitrary restrictions than with. This is, indeed, what happened with ALGOL 60.

6 Some problems with ALGOL 60

The syntactic generality of ALGOL 60 made it, decidedly, a much more elegant language than FORTRAN. The majority of its authors had a background in numerical mathematics. This shows in the absence of any inbuilt facilities to compute with texts, or with any other kind of structure than vectors and matrices of numbers. By the mid sixties, it was evident that this was a serious deficiency; non-numeric computing had become at least as important as numeric computation.

Some facility was needed by which programmers could add their own types. This required, of course, generalizing constructs to arbitrary types. In ALGOL 60 the types were given explicitly in the syntax rules. There was a rule for a conditional integer

5. More properly: to re-invent. Chomsky had before described context-free grammar in a linguistic context to characterize his "Type 2" languages (now generally known as context-free languages).

expression, and also for a conditional boolean expression, but not, for example, for a conditional string expression.

Another problem with ALGOL 60 had to do with type checking. The following is a program, written in ALGOL 60, that contains a type error:

```
begin procedure a(b, c, d, e, f, g);
        b(c, d, e, f, g, a);

        procedure s(t, u, v, w, x, y, z);
        z(s, t, u, v, w, x, y);

        a(a, a, a, a, a, s)
end
```

The procedure-call rule of ALGOL 60 results in the following call sequence:

```
a(a, a, a, a, a, s)
a(a, a, a, a, s, a)
a(a, a, a, s, a, a)
a(a, a, s, a, a, a)
a(a, s, a, a, a, a)
a(s, a, a, a, a, a)
s(a, a, a, a, a, a)
```

Here *s* is called with six parameters. But *s* requires seven parameters. In general there is no foolproof way to determine in advance whether an arbitrary ALGOL 60 program contains such a type error. This is rather obvious if we only consider "reachable code", since it is not decidable which parts of the program are reachable. One can take a more textual view and require that under repeated replacement of calls by expanded bodies there are no parameter mismatches. But even then the problem is undecidable, as was shown by Langmaack [2].

There are, of course, more shortcomings of ALGOL 60, in particular the conspicuous absence of input/output facilities. Adding these is, however, "merely" a matter of adding. It does not require a really new language. The problems mentioned above could not be solved without creating a new language.

7 From ALGOL 60 to ALGOL 68

Van Wijngaarden's insight was that the required generalization of the syntax rules could be obtained by introducing parametrized grammar rules [3]. Using BNF-like notation, and applying this to ALGOL 60 syntax (somewhat simplified), we see that the rules

⟨conditional boolean expression⟩ ::=
if ⟨condition⟩ ***then*** ⟨boolean expression⟩ ***else*** ⟨boolean expression⟩

⟨conditional integer expression⟩ ::=
if ⟨condition⟩ ***then*** ⟨integer expression⟩ ***else*** ⟨integer expression⟩

⟨conditional real expression⟩ ::=
if ⟨condition⟩ ***then*** ⟨real expression⟩ ***else*** ⟨real expression⟩

can be unified to a single rule

⟨conditional TYPE expression⟩ ::=
if ⟨condition⟩ ***then*** ⟨TYPE expression⟩ ***else*** ⟨TYPE expression⟩

if "TYPE" can stand for any of "boolean", "real" and "integer". The generalization requires now to describe what "TYPE" can stand for. This description forms the "metalevel" of the grammar. I do not know if Van Wijngaarden was aware of other varieties of two-level grammars, such as affix grammars or attribute grammars, but where these had either a limited or not grammatically specified domain for the metalevel, Van Wijngaarden grammars have for the metalevel a conventional context free grammar.

Just like context free grammar permitted to describe the recursive formation rules for program texts in ALGOL 60, here they allow to give recursive formation rules for the types[6] of ALGOL 68. And, just as for ALGOL 60, this makes it harder rather than easier to have exceptions.

Today it is entirely commonplace that the type system of a language has recursive formation rules. In 1968 it was an innovation; and precisely what is needed to allow the users to define a good interface between the implementation of their "abstract machine" and the implementation of the abstract algorithm.

The main criticism I see for the type system of ALGOL 68 is the low level of the type constructor "***ref***", in particular the coupling of assignability with referring[7]. For more detailed criticism, see Koster [1].

At least as important as the type system *per se* is the fact that ALGOL 68 is strongly typed.

8 After ALGOL 68

It is inevitable that any retrospective activity has some of the power of hindsight. ALGOL 68 was not only the brainchild of van Wijngaarden, but also a child of its time. With our present knowledge of the principles of programming languages, surely the current major programming languages do a much better job. Or do they?

It has been remarked that ALGOL 60 was an improvement over most of its

6. In ALGOL 68 idiom, the term "mode" is used for what is usually called "type". In this article I use the more common term "type".

7. This is especially severe since the obvious way to define "algebraic" types, with recursion through "***union***" and "***struct***", requires the use of "***ref***" for "shielding to **yin**". (see rules 7.4.1 of the Revised Report).

successors[8]. To make a joke in the style of van Wijngaarden, but reflecting my personal opinion: ALGOL 68 would have been an improvement over most of its successors, had it had any.

Is it true that the "implicit view of the programmer's task" underlying the design of ALGOL 68 was very much the same as, say, 1958? Whether this was so or not, ALGOL 68 introduced a view on types that makes programming easier. It is a view that is common now. Unfortunately, not much has been added since, at least not in major available languages. Although C owes much of its type system from ALGOL 68, it is clearly a step back towards the machine level. In particular, it is annoying — to put it extremely mildly — that the allocation and deallocation of dynamic "non-stack" storage is the responsibility of the C programmer, and that the semantics of C gives no support for keeping pointers in check.

A true improvement, in my opinion, is type polymorphism. I am more dubious about the object-oriented paradigm. I've seen no linguistic approach to that seems of an acceptable neatness to me.

9 Final remarks

There is a tension between two viewpoints concerning the relation between language design and program correctness. One viewpoint is that a good language makes it hard to write bad programs. The other viewpoint is that it makes it easy to write good programs. Van Wijngaarden was an outspoken adherent of the latter viewpoint; he considered the first one "paternalistic". Nevertheless, the design of ALGOL 68 is such that many if not most "clerical errors" may be statically detected, and this was already so in the version of 1968.

No language design can really prevent programmers to create an inextricable mess, and a language in which it is really hard to write bad programs probably also makes it hard to write good programs. Nevertheless, some things in ALGOL 68 are error prone (like "***ref***" mentioned before), and I think the language could have been better if more attention had been paid to such problems. The same is true for almost all languages that saw the light since.

For the record, I want to state that — although my name is associated with, in particular, the Revised Report [5] — I do not feel I have, personally, a stake in any evaluation of the design of ALGOL 68. My role has mainly been confined to "debugging" and polishing an existing description of an existing design, and this has had at most a marginal influence on the language as a programmer perceives it. I do feel responsible for remaining errors in the Revised Report (not counting the section on Transput declarations); in particular, being reminded of the painful fact that I overlooked that "**real field letter y integral field letter l**" contains "**yin**" (see rule 7.3.1.c of the Revised Report) still can make me blush.

8. I do not remember to whom this witticism should be ascribed, but Tony Hoare comes to mind as a plausible source. In any case I am pretty sure it was not Niklaus Wirth.

References

1. C.H.A. Koster (1976). The mode system in ALGOL 68. *New Directions in Algorithmic Languages 1975* (S.A. Schuman, ed.), 125-138, IRIA, Rocquencourt.
2. H. Langmaack (1973). On correct procedure parameter transmission in higher programming languages. *Acta Informatica* **2**, *110-142.*
3. A. van Wijngaarden (1965). *Orthogonal design and description of a formal language.* Mathematisch Centrum, Amsterdam, Mathematical Centre Report MR 76.
4. A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster (1968). *Final Draft Report on the Algorithmic Language ALGOL 68.* Mathematisch Centrum, Amsterdam, Mathematical Centre Report MR 100.
5. A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, Eds (1975). *Revised Report on the Algorithmic Language ALGOL 68.* Acta Informatica **5**, 1-236.