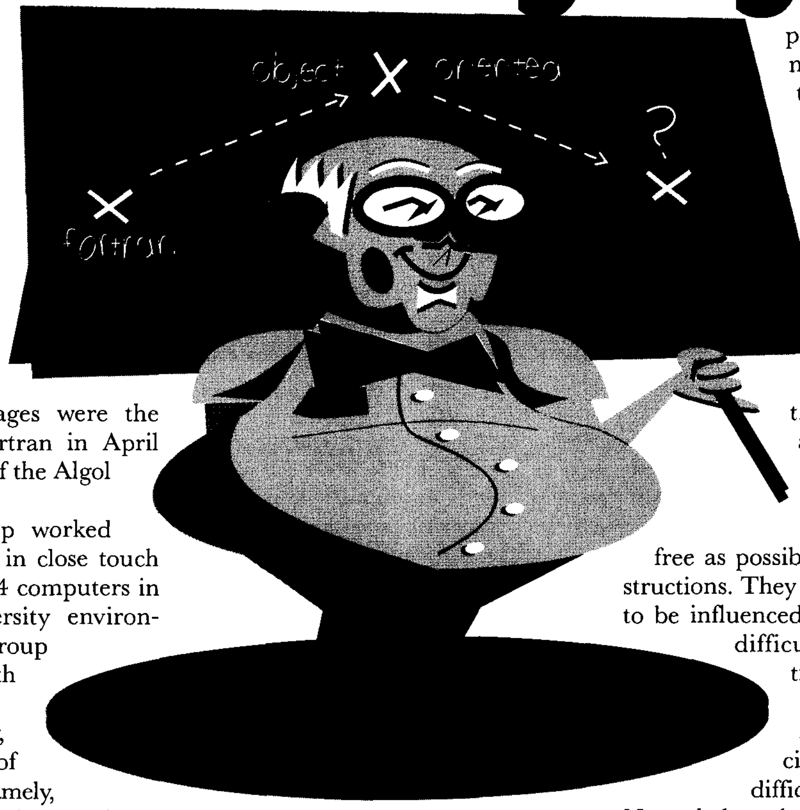
From Fortran and Algol to Object-Oriented Languages



In the latter part of 1952 and 1953, people began to experiment with symbolic programming systems that can now be recognized as the remote ancestors of modern high-level programming languages. However, the first major landmarks in the development of programming languages were the announcement of Fortran in April 1957 and the issuing of the Algol 60 report in 1960.

The Fortran group worked within IBM and was in close touch with users of IBM 704 computers in industrial and university environments. Although the group broke new ground, both in language design and in compiler technology, they never lost sight of their main objective, namely, to produce a product that would be acceptable to practical users with real problems to solve. In this they were very successful. There was, of course, much skepticism, and it would be wrong to say that users of IBM 704s across the U.S. switched from assembly language programming to Fortran overnight; however, many of those who were persuaded to give Fortran a try became enthusiastic.

Algol 60 was the work of an international group with members from Germany, Switzerland, the U.S., England, Denmark, etc. The spade work had been done as a result of an earlier initiative taken by a small group of mathematicians in Switzerland and Germany with a background in numerical analysis. Algol 58, as their language was called, was of short-lived importance. However, it did provide a starting point for the Algol 60 group which met for one week of intensive activity in Paris in 1960. At the end of the week they were able to agree on a report that stood the test of time.

The Algol group, although international in composition, was dominated by its continental members. In 1960, there were very few powerful computers on the continent of Europe, and these members, unlike their Fortran counterparts, were not used to struggling with the day-to-day problems of getting work through a computer center. Their interest was more theoretical. Their aim was to define a language based on coherent principles that would be as free as possible from ad hoc constructions. They were determined not to be influenced by implementation difficulties for any particular machine and, if a feature in the language made efficient implementation difficult, then so be it. Nevertheless, they had in mind a very clear implementation model, namely, that the language was to be implemented on a stack. All memory allocation was to take place on the stack. This model dominated the design of the language. The scope rules of Algol 60—by which a variable declared in a block could only be accessed in the block itself, or in blocks interior to it—naturally followed.

Algol 60 made little headway in the U.S. as a practical computing language, being taken up by only one major manufacturer, namely Burroughs. This was no doubt principally for the practical reasons I have mentioned. Another reason was that com-

Maurice V. Wilkes

ILLUSTRATION: DENNIS AHLGRIM

puter users generally were unprepared for the abstract sophistication of the Algol language. Most of them had been trained in other disciplines such as physics and engineering, rather than in pure mathematics. As far as computing went, they saw themselves as practical people solving real problems and felt a distrust for those whom they regarded as having their heads in the clouds. This attitude did not entirely disappear until the effect of computer science teaching in the universities made itself felt.

A particular feature that helped distance practical people from the Algol enthusiasts was recursion. On the advocacy of John McCarthy, who had only a few years before developed the Lisp language, the Algol group decided that all procedures should be recursive. This decision accorded well with their philosophy of implementation on a stack. The long-term rightness of this decision has become amply apparent, but at the time it was a concept that many people found difficult to understand; even when they had understood it, they could not see what good it was, and they feared it might do harm by leading to inefficiency.

When a change was made to an Algol program, recompilation of the whole program was necessary. This was a serious disadvantage especially at a time when most computers made use of slow punched card readers for input. It is a problem that has continued to plague block-structured languages ever since.

In Fortran, which had no block structure—and may therefore be described as a *flat* language—separate compilation presented no difficulty. The Fortran Monitor System for batch processing came into use about 1958 and made it especially easy to combine in one program parts written in Fortran with others written in assembly code. The Fortran Monitor System broke new ground, and may be regarded as the forerunner of modern operating systems. It was a major factor in promoting the general adoption of Fortran.

As time went on, Algol-type languages gradually became accepted for non-numerical programming, including system programming, and also to some extent for small- and medium-scale, numerical programming. However, people engaged in large-scale numerical computation—especially in nuclear and atomic physics—remained loyal to Fortran. This was partly because their programs were relatively simple in logical structure, and Fortran provided all they needed. Moreover, users of the Cray I computer found Fortran, with some special features added, enabled them to optimize their programs to take full advantage of vector hardware. Fortran, in the version known as Fortran 77, is still by far the most popular language for numerical computation, although C is making a small amount of progress, especially among the younger programmers.

Programming Languages as a Scientific Study

As a subject for study, Algol attracted wide-spread interest early on in the universities in the U.S. and elsewhere. In fact, a major achievement of the Algol pioneers was to create a new academic discipline, namely the scientific study of programming languages. This discipline had its practical side and its theoretical side. The theoreticians found challenging problems in syntax definition and syntax analysis; later, when the most important of these challenges had been successfully met, they moved to other computer-related areas, notably to complexity theory.

The block structure proved a fruitful idea; the use of a single stack provided a simple and automatic method of recycling memory no longer in use. However, it is unfortunate block structure continued for so long to dominate, to the exclusion of all other models, the thoughts of academic language specialists. It is in my view regrettable that a scientific study of flat languages such as Fortran was not made, and their merits and potentialities evaluated in the light of what had been learned from Algol.

As long as the stack remained the kingpin on which all memory allocation depended, there was no hope of advances in modularity or data hiding. The first sign of a break away from pure block structure was the appearance in advanced languages of the heap. Allocation of memory from the heap was an alternative to allocating space on the top of the stack. This paved the way for object-oriented programming which is, in my view, the most important development in programming languages that has taken place for a long time.

Technicalities apart, an object-oriented language provides the programmer with two major benefits, namely, modularity and data hiding. A program is composed of a number of modules; each module is self-contained, has a clear interface to other modules, and is capable of being separately compiled. The writer of a module can decide for himself the extent to which the data within it are to be protected from misuse or accidental corruption by being hidden from the outside world.

The introduction of a heap required a new policy for recovering memory no longer in use. It was at first assumed that an automatic garbage collector along the line of that used in Lisp would be essential. It is true that list processing in the Lisp sense would be very difficult without a garbage collector; however, experience has shown that for much work the programmer can get on quite well without one. For example, few implementations of C++ rejoice in a garbage collector. The programmer has to take care that leakage of memory does not become a problem.

The heap, with a garbage collector, and classes (a seminal concept on the way to object-oriented programming), were to be found in Simula 67. The importance of these features to programming language in general was only slowly realized. This was perhaps because Simula was offered to the world as a simulation language rather than as a general purpose language.

Object-oriented programming languages may still be described as being in a state of evolution. No completely satisfactory language in this category is yet available. Modula-3 is well designed, but it has so far attracted attention mainly in research circles and as a language for teaching. C++ may be described as an object-oriented language and can handle large problems, but it shows signs of its nonobject-oriented origins.

Compatibility

Designers of programming languages—whether individuals or committees—differ in their attitude to compatibility. How easy should they make it to convert a program written in an old language to the new language?

Recently, a new ISO standard for Fortran has been adopted. This is known as Fortran 90 and it is hoped that eventually it will come into general use to replace Fortran 77. The whole of Fortran 77 is included within Fortran 90 so existing programs will run without change.

It is harder to imagine a stricter view of compatibility. It at once creates a problem. Users who change over to Fortran 90 will presumably wish to make use of the new features, otherwise they will have no motivation to make the change. They will have to adapt their own style of programming, dropping Fortran 77 features now regarded as outmoded and embrace the newer facilities. The Fortran 90 standard gives some indications as to what those features are, but most users will need formal training if they are to use the language to best advantage.

Niklaus Wirth took a less strict view of compatibility when he designed the well-known language Pascal. This is a close relation of Algol 60, and it is easy for an Algol 60 programmer to learn; however, there was never any thought that a Pascal compiler might be able to accept Algol 60 programs. Wirth has designed a number of languages. Each represents his developing view of what a language should be. If you adopt one of Wirth's languages, then you must go along with his view of how computer programs should be written. He puts stress on elegance of structure, with few if any exceptions to the rules, and on the elimination of unnecessary or redundant features. His aim is to produce a "lean" language that can be described in a slim manual and is easy to learn.

C++ was designed by Bjarne Stroustrup who took as his starting point the language C. Stroustrup aimed at producing a language cleaner than C and one that would make object-oriented programming possible. He was not so much concerned with elegance in the mathematical sense. He was cautious of tailoring the language to his own, possibly limited, view of what would be good for users. In fact, he retained various constructs from C he did not particularly like in order to make sure he did not unintentionally create difficulties for people working in programming areas they understood better than he did.

Stroustrup remarked to me recently, "my impression is that most other language designers are less reluctant to impose their views on programmers and that some consider such imposition their duty." As a result, C++ does not impose any particular style of programming on the programmer. However, a designer who takes this view has the duty to give advice to programmers on how to develop an appropriate style of programming, and Stroustrup thoroughly accepts this duty.

If I had to draw any conclusion from these observations on current developments, it would be that the subject of programming languages, in spite of its 35 years of existence is still a very immature one. To be suitable for very large problems as well as for small ones, I do not believe a programming language has itself to be large and covered with warts. On the contrary, I believe one day we will have simple and powerful languages that do not force unnatural ways of working on the programmer. More experimentation is needed, but I am optimistic enough to hope we are more than halfway there. C

Maurice V. Wilkes received the ACM Turing Award in 1967 and is the author of Memoirs of a Computer Pioneer, *MIT Press, 1985.*

Eckert-Mauchly Award Nominations

Nominations are being solicited for the ACM/IEEE Eckert-Mauchly Award, which is presented annually to an outstanding computer architect for seminal contributions to computer and digital system architecture.

Questions regarding the award or nomination process should be directed to H. C. Torng, Eckert-Mauchly Award Committee, (607) 255-5191, torng@ee.cornell.edu.

The deadline for submitting nominations is October 4. Nominations statements, which must not exceed two pages, should include a proposed citation and a justification with published papers, patents and other material in the public domain.