Thirty Years of Programming Languages and Compilers

N. Wirth

Institut für Computersysteme, ETH Zürich

Norman Ramsey

Summary

The first part of this talk is a brief review of the development of high-level programming languages whose essential purpose is to let the programmer formulate programs in terms of abstractions, i.e. of structures independent from those offered by individual computers. We regard Algol 60 as the base of high-level languages. Languages designed by this author are mirrors of the continuous evolution of programming methodology during the subsequent decades: Pascal originated around 1970 and reflects the ideas of Structured Programming, Modula-2 (1980-) those of Modular Programming, and Oberon (1990) those of Object-oriented Programming. During these three epoches the subjects of program design and software engineering have made substantial advances.

The second part of the presentation starts with the question, whether today's programmers at large make use of this progress. The answer is, in view of the current spread of the language C, rather negative, because C must be equated due to its lack of guarded abstractions with low-level languages of the 60s rather than with high-level languages of the Algol-line. Particularly regrettable is the broad and uncritical acceptance of C in education, where the learning of method and abstraction, of choosing appropriate, structured representations, and of clear and concise formulation should be the primary goal.

In the last part, a number of suggestions are offered for reverting the current trend. Among them is reflection about what is truly essential in Computer Science education: *The influence of academic education on practice in industry and commerce.*

1. Introduction

Thirty years ago, in late 1962, I designed my first programming language, called Euler, as a true academic exercise in generalizing the concepts of Algol 60, which I considered fundamental for all programming. Afterwards, the subject continued to fascinate me, and, with a shift from mostly conceptual concerns to practical needs, I developed in succession PL 360, Algol W, Pascal, Modula-2, and Oberon. I therefore feel an urge to assess the developments of these thirty years, the evolution of languages and of compiler technology , focussing on procedural languages. They mirror most closely the fundamental characteristic of our computer, namely the store with individually updatable cells, and therefore have remained the most suitable notations to express large systems. If I refer to my afore-mentioned languages, it is because they reflect major milestones in the development of the programming discipline quite well through added features mirroring the new requirements imposed by new programming strategies. This presentation consists of three parts, concerned respectively with the past, the present, and the future.

2. The development of procedural languages and compiling techniques

In 1961, I was a member of a small group at UC Berkeley engaged in extending and reworking a compiler for the language NELIAC, a "dialect" of Algol 58, which – progressive for that time – was programmed in its own language. It taught me the value of the bootstrapping technique which is so fundamental in software design. The program was an eldorado for the hacker, but a nightmare for

the one seeking orderly development and the clarity of mathematical expression, which at that time was still believed to be a cornerstone of computer science.

My goal for research was clear: to introduce method into the obscure art of compilter design, and mathematical conciseness and precision into languages. A language evidently had to represent a higher level of abstraction of the basic features of computers. But NELIAC was far from that goal.

I vividly remember the impact of the appearence of the Algol 60 Report. The one most important innovation of Algol 60 was the definition of its syntax with mathematical rigor. It immediately suggested to disentangle those parts of a compiler which handle structural analysis of the source text from the parts that generate the target code. A flurry of research activities began in order to find efficient methods of syntactic analysis: parsing techniques.

The distinction between top–down and bottom–up methods was identified, and the recursive–descent and table–driven techniques were established. Floyd defined the notion of operator–precedence grammars, which we generalized into precedence grammars. Increasingly general (i.e. less restrictive) classes of grammars were postulated, together with increasingly sophisticated analysis algorithms capable of parsing languages falling into the given class: SLR, LALR, etc.. The activities culminated in Knuth's two landmark papers about top–down (LL) and bottom–up (LR) parsing [1, 2].

In retrospect, this concentration on the easier aspect of compiling was misguided and even counterproductive. It seduced language designers to ignore problems of syntax analysis, in the belief that future techniques would cope with any syntax. As a result, languages were defined with trivial mistakes in their syntax requiring complex algorithms for parsing, whereas a simple correction would have permitted the use of a much simpler and more efficient method.

In a similar vein, table–driven parsing suggested the idea of automatic compiler generation. Going one step further, we designed a compiler in 1966 for a language extensible on–the–fly, in which certain statements in the source code were directed to the parser, adding (or deleting) productions to (from) the grammar. These efforts both stimulated and depended on formalization of the other part of a compiler: code generation. The basic idea was to use the syntax of a language as the scaffolding, and to associate with each syntactic production a rule for code generation. A contribution in this direction was made by the specification of the Euler compiler [3] in 1966. The idea then led to the notion of attributed grammars, which still appear as a research topic today.

Inspite of Algol's elegance, its expressive power, on the other hand, was not adequate for the purpose of programming compilers and operating systems. In the meantime, increasingly powerful and complex languages were postulated and implemented, among them Algol W [4], PL/1, Algol 68 [5], Ada. Not only did they make it obvious that syntax analysis is a minor part of compilation, but none of these languages fully satisfied the needs of the system programmer either. Evidence of this fact is that their compilers were programmed in either assembler code or in a low–level, so–called system programming language, such as BCPL, PL360, PL/S, in essence representing syntax–sugared assembler codes. The few exceptions confirm the rule: Hoare had programmed an Algol compiler using (Elliott–) Algol, and the Algol compiler for the B5000 computer was expressed in (much) Extended Burroughs Algol.

The movement of *Structured Programming*, initiated by the so–called Software Crisis, found its reflection in the language Pascal [6]. It systematized the notions of control structures and extended structural design to the domain of data type definition. It was a welcome vehicle in the classroom due to its concise and systematic structure. But it was (before publication!) also implemented; its compiler was written in Pascal itself, demonstrating its suitability for systems programming. This in itself constituted a giant step forward: The compiler as well as subset compilers were distributed in

source code, making it possible to transport (reimplement) it on various computers with relative ease. The publication also provided a welcome case for studying a factual piece of software, a clear indication of the value of using a high-level language in software design. A similar success was Brinch Hansen's Solo system, described in Concurrent Pascal, in the field of operating systems [7].

Whereas Pascal had been born in the era of Structured Programming, Modula-2 [8] was a child of the era of *Modular Programming*. It had become evident that large systems could be developed and maintained only if they were designed in a modular fashion. The new construct was the module (unit) that could be compiled separately. *Separate compilation* must be understood in the context of strict static data typing. Compilers must guarantee type consistency not only among objects declared within a module, but also among objects defined in separately compiled modules. Separate compilation therefore is much more intricate than independent compilation, which was in practice since the advent of assemblers.

As in the case of Pascal, the new concept and its usefulness in the implementation of Modula compilers, which were, of course, expressed in Modula itself, were tested. The language had proven its value convincingly before the language report was published. The idea of separate modules with separate interface definitions stemmed from the Mesa project at Xerox PARC, and it truly constituted one of the rare breakthroughs in software engineering.

I believe in progress through evolution rather than revolution. Therefore, my latest language Oberon builds upon Modula-2. In my opinion, its most valuable contribution is the integration of object-oriented concepts with the established concepts and facilities of classical procedural programming. The essential and novel concept of subclassing, introduced by Simula 67 long before the term object-oriented was coined, is shown to correspond to the derivation of a new data type (T1) from an existing one (T0), where T1 is postulated as being compatible with T0 (but not vice-versa). The recognition that classes correspond to types, subclasses to extended types, methods to (type-bound) procedures, and objects to variables (instances of types) constitutes a significant conceptual simplification and clarification. Ironically, it led some experts to criticize Oberon as a non-object-oriented (and therefore antiquated) language, probably only because the typical object-oriented terminology was missing. We emphazise that object-oriented facilities and strong typing of all data are combined, permitting the programmer to take advantage of both object-oriented techniques and type-safe, modular programming.

Compiler technology was influenced during the 1980s by developments in hardware (i.e. semiconductor technology) more profoundly than by any other factor. I refer here less to the increase in speed than to the availability of large – by the standards of the 1970s huge – memories. Whereas before 1980 enormous efforts were necessary to cram compilers into the computer's memory, such efforts quickly became superfluous and detrimental. Multipass-compilers with their cumbersome sequentializers and intermediate code representations became obsolete. This is a good example of how hardware abundance can make software tasks *simpler* and much more efficient.

Another, equally important influence of memories was the shift of priorities from density of generated code to speed. It is manifest in the transition from CISC (complex) to RISC (reduced instruction set) computers. RISC computers part from instructions with complex interpretations (time-consuming decoding) and involved addressing modes (possibly involving several memory accesses), and retain only simple instructions executable in a single clock cycle. This implies another simplification of compilers, because complicated algorithms for finding the constellations where a complex composite instruction is applicable can be discarded. The cost is a larger number of (simpler) instructions, resulting in increased code length, i.e. smaller code density. However, large memories and caches (speeding up sequential access) make this increase easily acceptable. Indeed, compiler construction has become simpler. But other requirements appeared, as we shall see shortly,

partially offsetting this gain.

3. Where do we stand today?

Undeniably, language design and compiler technology have undergone tremendous improvements in the past thirty years. Whereas hundreds of programmers used to labour for years building a compiler or an operating system, nowadays small teams achieve the same goals in a single year or less. I can muster not a single bit of nostalgia for the languages and compilers of thirty years ago.

Are these advances genuinely reflected in the state of the software industry in 1992? Is its productivity any greater than in 1962? Are its products of better quality, more efficient, more reliable, more economical? Are compilers and operating systems more systematic, less bulky? Are language specifications, instruction manuals more concise, better structured, more readily comprehensible? And specifically: Do programmers at large make use of structured disciplines and high-level languages supporting them in their effort to produce systematic, structured, modular, reliable, trustworthy software? Not really!

Today, programmers at large program in C. And where is the place of C in our chronicle of progress of the last thirty years? Right at the beginning, because C offers a very low level of abstraction. In this respect, we must place it among language like NELIAC (1959). BCPL (1962), and PL360 (1965). It is devoid of those fundamental notions of typed, structured data, of static type checking, of modules with checkable interface specifications, of mathematical clarity. An admirable example of the programming style inspired by C is the following piece of code which I recently encountered:

```
char *s1, APs1[maxlen];
for(i = 0; AP(s1+i) >= 0; i++) APs1[i] = F(s1+i);
if (i >= maxlen) error
```

Its equivalent text formulated in Pascal, exhibiting the mistake succinctly, is

```
VAR i, s1: INTEGER;
   APs1: ARRAY [0 .. maxlen] OF INTEGER;

i := 0;
WHILE F(s1 + i) >= 0 DO
   BEGIN APs1[i] := F(s1+i); i := i+1 END ;
IF i >= maxlen THEN error
```

In fact, C represents assembler code under the coating of an unattractive syntax fraught with cryptic symbols. The conclusion that we regrettably draw is that the software industry is either unwilling or incapable of taking advantage of much of the progress achieved. The sad fact is that the cost of this behaviour is passed on to the industry's customers.

We might swallow this and turn to more important matters. What makes me really sad, however, is the influence of the C-wave on education in computer programming, and in particular the negligence with which those who ought to know what really matters succumb, the educators. I conclude that the educators' attitude towards programming is unchanged from (or has reverted to) that of thirty years ago. It seems that an eldorado for hacking is more desirable than a disciplined notation for expressing trustworthy and provable algorithms and data structures. After all, programs are still mostly concocted in order to "run" rather than to be comprehensible, a state of affairs that was already recorded and deplored in 1962 [9]. In those years *Communications of the ACM* used to contain a section entitled Algorithms. Algorithms were published in Algol 60 as a standard publication language. Today, many articles contain pieces of C-programs in forms difficult to understand and of questionable value for dissemination. Also in this respect: thirty years undone!

I remember well the difficulties experienced in propagating the idea of a high-level language (Pascal) in the industrial world. The everpresent argument for retaining assembler code and Fortran was efficiency. Valiant efforts in compiler optimization finally made Pascal-generated code competitive. Nowadays, the argument of efficiency is being reserved for hardware, which indeed becomes faster each year. The increase is so predominant that when concerned with software effectiveness one runs the danger of being regarded as old-fashioned. On many occacions I have witnessed the confirmation of Reiser's law: Software is getting slower faster than hardware is getting faster. We conclude that the progress of hardware technology has at least one detrimental effect: it makes poorly designed software acceptable.

Let me support this statement by an example. A compiler has been developed for a modern language in an industrial laboratory for a language more complex but in expressive power comparable with Oberon. The compilation of the compiler by itself takes 40 min. On the same computer, the compilation of the Oberon compiler by itself takes 6 s. The slow compiler was nevertheless adopted and justified considering the fact that tomorrow's computers will make compilation times negligible. One of the reasons for the compiler's alarming ineffectiveness: it compiles to C instead of directly into machine code (the C-compilations are included in the given figure). Let me add that such a monstrous design would have been quite impossible 30 (even 10) years ago, because no one would have had the patience to wait for the completion of a compilation for a few days or weeks. Thirty years undone?

In another case, a large number of workstations had to be retrofitted with additional memory, because 32 Megabytes were no longer sufficient to hold the system and still provide space for applications. In our own school, Sun workstations acquired a few years ago are being replaced by more modern models, because the manufacturer no longer supports that version of the operating system, and the new one does not fit the memory of the older computers. Again, the cost of these dubious developments are passed on to the - unfortunately uncritical and sadly helpless - consumer.

What has gone wrong? Where have we failed? I believe that we educators have failed in several respects. Mainly, software development has not been taught as a serious engineering activity. Instead, emphasis was put on computer "science". Let me list a few areas in which researchers have invested their time and energy in the pursuit of "science".

1. Syntax analysis, parsing methods. The subject has enjoyed its prime time between 1965 and 1975, but it still attracts attention today, and ever increasing sophistication is introduced and taught. This is witnessed by textbooks devoting hundreds of pages to the topic. On the other hand, I have developed the fastest compilers, using the simple recursive descent technique known since 1960, for Pascal, Modula, and Oberon. One cannot entirely suppress the impression that the fascination of theory has distracted from relevant issues.

2. Compiler compilers, automated code generation. Efforts in this area date back to 1965 and were made mostly with the goal of formalizing not only syntax but also semantics. A tabular representation of semantics in terms of code sequences or code generation rules associated with the syntatic productions was believed to be highly desirable. The need for cross-statement code optimization, however, makes these techniques rather unsatisfactory in practice. More important, however, is the question whether automated compiler generation is of practical relevance. Knowing how rarely one needs a new compiler, and how intricate the task of designing generators for efficient code is, I tend to deny this goal much relevance. It is much more realistic to construct compilers in a way that makes them portable, i.e. in a way which reduces the effort of writing code generators for different target architectures. The key to this goal is a proper modularization.

3. Compiler optimization. Enormous efforts have been spent on this topic, sometimes with questionable results. In many cases, the amount of labour was incommensurate with the result, and often the choice of a better source language or an adequate code architecture would have made the project superfluous. Concerning compiler optimization, it is essential to distinguish between the improvement of code for constructs where the programmer could have achieved the same effect by a different formulation of the program, and constructs for which this is impossible. An example of the former kind is the detection of common expressions, or of extracting expressions from loops in which their values remain unchanged. I have always regarded this as a misguided task, encouraging – if anything – sloppy programming practice. An example of the second kind is the opimization of addressing, e.g. for indexed addressing in sequential scans, or the optimal use of resources hidden from the programmer, such as registers. This is a highly relevant topic and is central to modern compiler technology, particularly since the advent of RISC architectures with a sizeable number of fast registers.

4. Language design. This research topic appears to continue to be attractive and has gained appeal through the new paradigms of functional, logic, and object–oriented programming. The majority of the new designs suffer from the same syndrome that made many languages of the past unsuccessful: their designers' love of complexity. Science has in the past made its significant steps forward through simplifications, integration of related concepts, systematization. The tendency to add features, which predominates in our field, goes in the opposite direction.

5. Standardization. The amount of effort invested in the subject of standardization of programming languages is truly mind–boggling. Particularly so, if the investment is compared with the outcome. I cannot explain this phenomenon, but believe that the standardization efforts were based on various misconceptions. The first one is that everything needs to be well–defined. For example, the omission of a definition of the terminating value of the control variable in a for–statement is considered intolerable. However, the language designers might have omitted this definition intentionally in order to provide more freedom for implementors. After all, the assumption that a professional programmer would not make use of absent definitions is certainly reasonable.

The second misconception is that formalization equals precision and rigor. Formal definitions of languages have turned out to be huge documents readable only by computers, and quite indigestible by human beings. I vividly remember my stupefaction at the sight of a definition of Modula–2's empty statement, a formal definition extending over a full page!

The third misconception is that standardization activities are appropriate occasions to extend a given language, or even to define another language. The inevitable result is a document in which neither industry nor academia can detect any reward, not the least because it becomes outdated before its appearance.

Summarizing, I conclude that computer scientists have spent too much effort on fascinating exercises and on maximizing publications, and too little on the difficult problems of genuine, practical relevance. This situation brings back to memory an apparently clairvoyant paper by Knuth containing further tales about the relationship between theory and practice [10], containing examples where theoretical efforts led to programs with inferior effectiveness when applied in practical circumstances. Today a reorientation is surely necessary to reestablish the rapidly dwindling influence of computer "science" to computing practice, which Parnas predicts to lead toward the disappearance of this discipline [11].

4. What should we do?

And this leads us to the final part of my talk, the question of how this course might be changed. Undoubtedly, educators are the ones who must make the first move. Educators are the ones who should teach future generations how to distinguish between what is essential and what is ephemeral, between what is lasting and what is of passing interest.

Perhaps it is just this distinction which many fail to emphasize or even recognize. The avalanche of novelties so loudly advertised in order to conceal their short life time has robbed many of the vision for what might be of lasting importance.

A fine example is the presumed importance of the exact imitation on the computer screen of one's messy desk–top laden with stacked–up documents. It is not technical merits, but rather convincive salesforces which have elevated the desk top metaphor into a de–facto standard – and a disinterest in evaluations of its cost vs. merit factor. Surely, neither the forceful salesman nor the lulled consumer has an inkling of the complexity behind the scenes. But at least educated computer scientists should become alerted and suspicious. Yet no one does! Perhaps – and fortunately for the salesmen – they have no inkling either.

It would of course be highly presumptuous to claim that there is a recipe, a panacea, a medicine for quick remedy, let alone to claim to know it. Let me therefore modestly offer a few recommendations that I believe would contribute towards a reorientation of our attitudes.

The first recommendation is to become *intolerant against mediocre software.* Mediocrity may have several faces; the worst being unreliability and incorrectness, leading to break–downs or wrong results. Too often have users accepted such deficiencies and learned to "program around the known bugs". Another face is user–unfriendliness in the form of an unsystematic user interface described by wishy–washy examples in place of concise rules. This is a sure symptom that also the system's internal structure suffers from ad–hoc solutions. And the third face is bulkiness, which often nobody cares to notice as long as the computer's memory is large enough.

And this leads to the second recommendation: when using and also when developing software, employ comparative measures for size and speed of your programs. With appalling frequency, programs are never compared with competitors, because this would be too cumbersome.

The third recommendation is a corollary of the second: when your software is too slow, do not merely buy a faster computer. In all likelihood, the software designer could do a better job, or you can find a better product. In fact, the faster computer will help to conceal ineffectiveness of one's development, not only from customers, but from the developer too. As Brooks aptly pointed out, the computer scientist is primarily a tool smith [12]. Tool smiths are judged by the tools they produce, not by the equipment they have acquired to develop them.

This leads me to the fourth recommendation: Use your tools yourself. First of all, if the tool is valuable, you are the first to profit and to bootstrap your own capabilities. And if the tool is inadequate, you are the first to notice, and you are given the chance to remedy the deficiencies before the client becomes aware of them.

My last recommendation is probably the most important of them, and it is addressed to educators: Emphasize and demonstrate the value of *clarity and abstraction.* Abstraction is the only forceful means to conquer complexity. Teach that programs must be written for human readers, and not only to be "run" by computers. If computer science education is worth anything, programs constructed by a professional should be distinguishable from those produced by a hacker.

The concepts and tools to realize these goals – high-level languages and effective compilers – are available today. Let us take advantage of them. And let us prove that the cynic was wrong who claimed that the only thing one can learn from history is that people do not learn from history.

References

1. D. E. Knuth. On the Translation of Languages from Left to Right. *Information and Control, 8,* 6, (Dec. 1965) 607 – 639.
2. D. E. Knuth. Top-Down Syntax Analysis. *Acta Informatica, 1,* 79 – 110 (1971).
3. N. Wirth. A Generalization of ALGOL, and its Formal Definition. *Comm. ACM 9,* 1, 13 – 23 and 89 – 99 (Jan. and Feb. 1966)
4. N. Wirth. A Contribution to the Development of ALGOL. *Comm. ACM, 9,* 6, 413 – 432 (June 1966)
5. A. van Wijngaarden et al. (eds.). Revised Report on the Algorithmic Language Algol 68. *Acta Informatica, 5,* 1 – 236 (1975).
6. N. Wirth. The Programming Language Pascal. *Acta Informatica, 1,* (1971), 35 – 63.
7. P. Brinch Hansen. The Solo Operating System. *Software – Practice and Experience, 6,* 2, 141 – 205 (1976).
8. N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.
9. E. W. Dijkstra. Some Meditations on Advanced Programming. Proc. IFIP-Congress 1962. North-Holland Publ.
10. D. E. Knuth. The Dangers of Computer-Science Theory, 1971.
11. D. L. Parnas. Education for Computing Professionals. *IEEE Computer,* Jan. 1990, 17 – 22.
12. F. P. Brooks, Jr. The Computer "Scientist" as Toolsmith. *Information Processing 77* (IFIP-Congress). B. Gilchrist, Ed., 625 – 634.

Paper presented at the *International Workshop on Compiler Construction (CC92),* Paderborn, Germany, Oct. 5–7, 1992.