Recollections about the Development of Pascal

N. Wirth
Institut für Computersysteme, ETH Zürich
CH–8092 Zürich

Abstract

Pascal was defined in 1970 and, after a slow start, became one of the most widely used languages in introductory programming courses. This article first summarises the events leading to Pascal's design and implementation, and then proceeds with a discussion of some of the language's merits and deficiencies. In the last part, developments that followed its release are recounted. Its influence chiefly derived from its being a vehicle for structured programming and a basis for further development of languages and for experiments in program verification.

Contents

Early History

The programming language Pascal was designed in the years 1968/69, and I named it after the French philosopher and mathematician, who in 1642 designed one of the first gadgets that might truly be called a digital calculator. The first compiler for Pascal was operational in early 1970, at which time the language definition also was published [Wirth, 1970].

These facts apparently constitute the anchor points of the history of Pascal. However, its genuine beginnings date much further back. It is perhaps equally interesting to shed some light on the events and trends of the times preceding its birth, as it is to recount the steps that led to its widespread use. I shall therefore start with a – more or less chronological – narrative of the early history of Pascal.

In the early 1960s, there existed two principal scientific languages: Fortran and Algol 60 [Naur, 1963]. The former was already in wide use and supported by large computer manufacturers. The latter – designed by an international committee of computing experts – lacked such support, but attracted attention by its systematic structure and its concise, formalized definition. It was obvious that Algol deserved more attention and a wider field of applications. In order to achieve it, Algol needed additional constructs making it suitable for purposes other than numerical computation. To

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
HOPL-II/4/93/MA,USA
© 1993 ACM 0-89791-571-2/93/0004/0333...$1.50

this end, IFIP established a Working Group with the charter of defining a successor to Algol. There was hope that the undesirable canyon between scientific and commercial programming, by the mid 1960s epitomized as the Fortran and Cobol worlds, could be bridged.

I had the privilege of joining Working Group 2.1. in 1964. Several meetings with seemingly endless discussions about general philosophies of language design, about formal definition methods, about syntactic details, character sets, and the whole spectrum of topics connected with programming revealed a discouraging lack of consensus about the approach to be taken. However, the wealth of ideas and experience presented also provided encouragement to coalesce them into a powerful ensemble.

As the number of meetings grew, it became evident that two main factions emerged from the roughly two dozen members of the Working Group. One party consisted of the ambitious members, unwilling to build upon the framework of Algol 60, unafraid of constructing features that were largely untried and whose consequences for implementors remained a matter of speculation, and eager to erect another milestone similar to the one set by Algol 60. The opponents were more pragmatic. They were anxious to retain the body of Algol 60 and to extend it with well-understood features widening the area of applications for the envisaged successor language, but retaining the orderly structure of its ancestor. In this spirit, the addition of basic data types for double precision real numbers and for complex numbers was proposed, as well as the record structure known from COBOL, the replacement of Algol's call-by-name with a call-by-reference parameter, and the replacement of Algol's overly general for-statement by a restricted but more efficient version. They hesitated to incorporate novel, untested ideas into an official language, well aware that otherwise a milestone might easily turn into a millstone.

In 1965, I was commissioned to submit a proposal to the WG which reflected the views of the pragmatists. In a meeting in October of the same year, however, a majority of the members favoured a competing proposal submitted by A. van Wijngaarden, former member of the Algol 60 team, and decided to select it as the basis for Algol X in a meeting of Warsaw in the fall of 1966 [van Wijngaarden, 1969]. Unfortunately, but as foreseen by many, the complexity of Algol 68 caused many delays, with the consequence that at the time of its implementation in the early 70s, many users of Algol 60 had already adopted other languages [Hoare, 1980].

I proceeded to implement my own proposal in spite of its rejection, and to incorporate the concept of dynamic data structures and pointer binding suggested by C.A.R. Hoare. The implementation was made at Stanford University for the new IBM 360 computer. The project was supported by a grant of the US National Science Foundation. The outcome was published and became known as Algol W [Wirth, 1966]. The system was adopted at many universities for teaching programming courses, but the language remained confined to IBM 360/370 computers.

Essentially, Algol W had extended Algol 60 with new data types representing double precision floating-point and complex numbers, with bit strings and with dynamic data structures linked by pointers. In spite of pragmatic precautions, the implementation turned out to be rather complex, requiring a run-time support package. It failed to be an adequate tool for systems programming, partly because it was burdened with features unnecessary for systems programming tasks, partly

because it lacked adequately flexible data structuring facilities. I therefore decided to pursue my original goal to design a general-purpose language without the heavy constraints imposed by the necessity of finding a consensus among two dozen experts about each and every little detail. The past experience had given me a life-long mistrust in the products of committees, where many participate in debating and decision making, and few perform the actual work made difficult by the many.

In 1968, I assumed a professorship at the Federal Institute of Technology in Zürich (ETH), where Algol 60 had been the language of choice among researchers in numeric computation. The acquisition of CDC computers in 1965 (and even more so in 1970), made this preference hard to justify, because Algol compilers for these computers were rather poorly designed and could not compete with their Fortran counterparts. Furthermore, the task of teaching programming – in particular systems programming – appeared highly unattractive, given the choice between Fortran and assembler code as the only available tools. After all, it was high time to not only preach the virtues of Structured Programming, but to make them applicable in actual practice by providing a language and compilers offering appropriate constructs. The discipline of Structured Programming had been outlined by E. W. Dijkstra [Dijkstra 1966] and represented a major step forward in the battle against what became known as the Software Crisis. It was felt that the discipline was to be taught at the level of introductory programming courses, rather than as an afterthought while trying to retrain old hands. This insight is still valid today. Structured Programming and Stepwise Refinement [Wirth 1971a] marked the beginnings of a methodology of programming and became a cornerstone in letting program design become a subject of intellectual respectability.

Hence, the definition of a new language and the development of its compiler were not a mere research project in language design, but rather a blunt necessity. The situation was to recur several times in the following decades, when the best advice was: Lacking adequate tools, build your own! In 1968, the goals were twofold: The language was to be suitable for expressing the fundamental constructs known at the time in a concise and logical way, and its implementation was to be efficient and competitive with existing Fortran compilers.

The latter requirement turned out to be rather demanding, given a computer (the CDC 6000) that was designed very much with Fortran in mind. In particular, dynamic arrays and recursive procedures appeared as formidable obstacles, and were therefore excluded from an initial draft of the language. The prohibition of recursion was a mistake and was soon to be rectified, in due recognition that it is unwise to be influenced severely by an inadequate tool of a transitory nature.

The task of writing the compiler was assigned to a single graduate student (E. Marmier) in 1969. As his programming experience was restricted to Fortran, the compiler was to be expressed in Fortran, with its translation into Pascal and subsequent self-compilation planned after its completion. This, as it turned out, was another grave mistake. The inadequacy of Fortran to express the complex data structures of a compiler caused the program to become contorted and its translation amounted to a redesign, because the structures inherent in the problem had become invisible in the Fortran formulation. The compiler relied on syntax analysis based on the table-driven bottom-up (LR) scheme adopted from the Algol-W compiler. Sophistication in syntax analysis was very much in style in the 1960s, allegedly because of the power and flexibility required to process high-level languages.

It occurred to me then that a much simpler and more perspicuous method could well be used, if the syntax of a language was chosen with the process of its analysis in mind.

The second attempt at building a compiler therefore began with its formulation in the source language itself, which by that time had evolved into what was published as Pascal in 1970 [Wirth, 1970]. The compiler was to be a single-pass system based on the proven top-down, recursive-descent principle for syntax analysis. We note here that this method was eligible because the ban against recursion had been lifted: recursivity of procedures was to be the normal case.

The team of implementors consisted of U. Ammann, E. Marmier, and R. Schild. After the program was completed – a healthy experience in programming in an unimplemented language! – Schild was banished to his home for two weeks, the time it took him to translate the program into an auxiliary, low-level language available on the CDC computer. Thereafter, the bootstrapping process could begin [Wirth, 1971b].

This compiler was completed by mid 1970, and at this time the language definition was published. With the exception of some slight revisions in 1972, it remained stable thereafter. We began using Pascal in introductory programming courses in late 1971. As ETH did not offer a computer science program until ten years later, the use of Pascal for teaching programming to engineers and physicists caused a certain amount of controversy. My argument was that the request to teach the methods used in industry, combined with the fact that industry uses the methods taught at universities, constitutes a vicious circle barring progress. But it was probably my stubborn persistence rather than any reasoned argument that kept Pascal in use. Ten years later, nobody minded.

In order to assist in the teaching effort, Kathy Jensen started to write a tutorial text explaining the primary programming concepts of Pascal by means of many examples. This text was first printed as a Technical Report and thereafter appeared in Springer-Verlag's Lecture Notes Series [Jensen, 1974].

The Language

The principal role of a language designer is that of a judicious collector of features or concepts. Once these concepts are selected, forms of expressing them must be found, i.e. a syntax must be defined. The forms expressing individual concepts must be carefully molded into a whole. This is most important, as otherwise the language will appear as incoherent, as a skeleton onto which individual constructs were grafted, perhaps as after-thoughts.

Sufficient time had elapsed that the main flaws of Algol were known and could be eliminated. For example, the importance of avoiding ambiguities had been recognised. In many instances, a decision had to be taken whether to solve a case in a clearcut fashion, or to remain compatible with Algol. These options sometimes were mutually exclusive. In retrospect, the decisions in favour of compatibility were unfortunate, as they kept inadequacies of the past alive. The importance of compatibility was overestimated, just as the relevance and size of the Algol-60 community had been. Examples of such cases are the syntactic form of structured statements without closing symbol, the way in which the result of a function procedure is specified (assignment to the function identifier), and the incomplete specification of parameters of formal procedures. All these deficiencies were

later corrected in Pascal's successor language Modula-2 [Wirth, 1982]. For example, Algol's ambiguous conditional statement was retained. Consider

```
IF p THEN IF q THEN A ELSE B
```

which can, according to the specified syntax, be interpreted in the following two ways:

```
IF p THEN [ IF q THEN A ELSE B ]
IF p THEN [ IF q THEN A ] ELSE B
```

Pascal retained this syntactic ambiguity, selecting, however, the interpretation that every ELSE be associated with the closest THEN at its left. The remedy, known but rejected at the time, consists of requiring an explicit closing symbol for each structured statement, resulting in the two distinct forms for the two cases as shown below:

```
IF p THEN                      IF p THEN
    IF q THEN A ELSE B END         IF q THEN A END
END                            ELSE B
                               END
```

Pascal also retained the incomplete specification of parameter types of a formal procedure, leaving open a dangerous loophole for breaching type checks. Consider the declarations

```
PROCEDURE P (PROCEDURE q);
BEGIN q(x, y) END ;

PROCEDURE Q (x: REAL);
BEGIN ... END ;
```

and the call P(Q). Then q is called with the wrong number of parameters, which cannot in general be detected at the time of compilation.

In contrast to such concessions to tradition stood the elimination of conditional expressions. Thereby the symbol IF clearly becomes a marker of the beginning of a statement, and bewildering constructs of the form

```
IF p THEN x := IF q THEN y ELSE z ELSE w
```

are banished from the language.

The baroque for-statement of Algol was replaced by a tamed version which is efficiently implementable, restricting the control variable to be a simple variable and the limit to be evaluated only once instead of before each repetition. For more general cases of repetitions, the while statement was introduced. Thus it became impossible to formulate misleading, non-terminating statements as for example

```
FOR i := 0 STEP 1 UNTIL i DO S
```

and the rather obscure formulation

```
FOR i := n–1, i–1 WHILE i > 0 DO S
```

could be expressed more clearly by

```
i := n;
WHILE i > 0 DO BEGIN i := i–1; S END
```

The primary innovation of Pascal was to incorporate a variety of data types and data structures, similar to Algol's introduction of a variety of statement structures. Algol offered only three basic data types, namely integers, real numbers, and truth values, and the array structure; Pascal introduced additional basic types and the possibility to define new basic types (enumerations, subranges), as well as new forms of structuring: record, set, and file (sequence), several of which had been present in COBOL. Most important was of course the recursivity of structural definitions, and the consequent possibility to combine and nest structures freely.

Along with programmer–defined data types came the clear distinction between type definition and variable declaration, variables being instances of a type. The concept of strong typing – already present in Algol – emerged as an important catalyst for secure programming. A type was to be understood as a template for variables specifying all properties that remain fixed during the time–span of a variable's existence. Whereas its value changes (through assignments), its range of possible values remains fixed, as well as its structure. This explicitness of static properties allows *compilers to verify whether rules governing types are respected. The binding of properties* to variables in the program text is called *early binding* and is the hallmark of high–level languages, because it gives clear expression to the intention of the programmer, unobscured by the dynamics of program execution.

However, the strict adherence to the notion of (static) type led to some less fortunate consequences. We here refer to the absence of *dynamic arrays*. These can be understood as static arrays with the number of elements (or the bounding index values) as a parameter. Pascal did not include parametrised types, primarily for reasons of implementation, although the concept was well understood. Whereas the lack of dynamic array variables may perhaps not have been too serious, the lack of dynamic array parameters is clearly recognised as a defect, if not in the view of the compiler designer, then certainly in the view of the programmer of numerical algorithms. For example, the following declarations do not permit procedure P to be called with x as its actual parameter:

```
TYPE   A0 = ARRAY [1 .. 100] OF REAL;
       A1 = ARRAY [0 .. 999] OF REAL;
VAR    x: A1;
PROCEDURE P(x: A0); BEGIN ... END
```

Another important contribution of Pascal was the clear conceptual and denotational separation of the notions of structure and access method. Whereas in Algol W arrays could only be declared as static variables and hence only be accessed directly, record structured variables could only be accessed via references (pointers), i.e. indirectly. In Pascal, all structures can be either accessed directly or via pointers, indirection being specified by an explicit dereferencing operator. This separation of concerns was known as "orthogonal design", and was pursued (perhaps to extreme) in Algol 68.

The introduction of explicit pointers, i.e. variables of pointer type, was the key to a significant widening of the scope of application. Using pointers, dynamic data structures can be built, as in list-processing languages. It is remarkable that the flexibility in data structuring was made possible without sacrificing strict static type checking. This was due to the concept of *pointer binding*, i.e. of declaring each pointer type as being bound to the type of the referenced objects, as proposed by [Hoare, 1972]. Consider, for instance, the declarations

```
TYPE  Pt = ↑ Rec;
      Rec = RECORD x, y: REAL END ;
VAR   p, q: Pt;
```

Then p and q, provided they had been properly initialized, are guaranteed to hold either values referring to a record of type *Rec*, or the constant NIL. A statement of the form

```
p↑.x := p↑.y + q↑.x
```

turns out to be as type-safe as the simple x := x + y.

Indeed, pointers and dynamic structures were considerably more important than dynamic arrays in all applications except numeric computation. Intricately connected to pointers is the mechanism of storage allocation. As Pascal was to be suitable as a system-building language, it tried not to rely on a built-in run-time garbage collection mechanism as had been necessary for Algol W. The solution adopted was to provide an intrinsic procedure NEW for allocating a variable in a storage area called the heap, and a complementary one for deallocation (DISPOSE). NEW is trivial to implement, and DISPOSE can be ignored, and indeed it turned out to be wise to do so, because system procedures depending on programmer's information are inherently unsafe. The idea of providing a garbage collection scheme was not considered in view of its complexity. After all, the presence of local variables and of programmer-defined data types and structures require a rather sophisticated and complex scheme crucially depending on system integrity. A collector must be able to rely on information about all variables and their types. This information must be generated by the compiler and, moreover, it must be impossible to invalidate it during program execution.

The subject of parameter passing methods had already been a source of endless debates and hassles in the days of the search for a successor to Algol 60. The impracticality of its name parameter had been clearly established, and the indispensability of the value parameter was generally accepted. Yet there were valid arguments for a reference parameter, in particular for structured operands on the one hand, and good reasons for result parameters on the other. In the former case the formal parameter constitutes a hidden pointer to the actual variable, in the latter the formal parameter is a

local variable to be assigned to the actual variable upon termination of the procedure. The choice of the reference parameter (in Pascal called VAR-parameter) as the only alternative to the value parameter turned out to be simple, appropriate, and successful.

And last but not least, Pascal included statements for input and output, whose omission from Algol had been a source of continuing criticism. Particularly with regard to Pascal's role as a language for instruction, a simple form of such statements was chosen. Their first parameter designates a file and, if omitted, causes the data to be read from or written to the default medium, such as keyboard and printer. The reason for including a special statement for this purpose in the language definition, rather than postulating special, standard procedures, was the desire to allow for a variable number and different types of parameters:

```
Read(x, y); ... ; Write(x, y, z)
```

As mentioned before, a language designer collects frequently used programming constructs from his own experience, from the literature, or from other languages, and molds them into syntactic forms in such a way that they together form an integrated language. Whereas the basic framework of Pascal stems from Algol W, many of the new features emerged from suggestions made by C.A.R. Hoare, including enumeration, subrange, set, and file types. The form of COBOL-like record types was due to Hoare, as well as the idea to represent a computer's "logical words" by a well-known abstraction, namely sets (of small integers). These "bits and pieces" were typically presented and discussed during meetings of the IFIP Working Group 2.1 (Algol), and thereafter appeared as communications in the Algol Bulletin. They were collected in his *Notes on Data Structuring* [Hoare, 1972].

In Pascal, they were distilled into a coherent and consistent framework of syntax and semantics, such that the structures were freely combinable. Pascal permits the definitions of arrays of records, records of arrays, arrays of sets, and arrays of record with files, just to name a few possibilities. Naturally, implementations would have to impose certain limits to the depth of nesting due to finite resources, and certain combinations, such as a file of files, might not be accepted at all. This case may serve as an example of the distinction between the general concepts defined by the language, and supplementary, restrictive rules governing specific implementations.

Although the wealth of data types and structuring forms was the principal asset of Pascal, not all of the components were equally successful. We keep in mind that success is a subjective quality, and opinions may differ widely. I therefore concentrate on an "evaluation" of a few constructs where history has given a reasonably clear verdict. The most vociferous criticism came from Habermann, who correctly pointed out that Pascal was not the last word on language design. Apart from taking issue with types and structures being merged into a single concept, and with the lack of constructs like conditional expressions, the exponentiation operator, and local blocks, which were all present in Algol 60, he reproached Pascal for retaining the much-cursed go to statement [Habermann, 1973]. In hindsight, one cannot but agree; at the time, its absence would have deterred too many people from trying to use Pascal. The bold step of proposing a goto-less language was taken ten years later by Pascal's successor Modula-2, which remedied many shortcomings and eliminated several

remaining compatibility concessions to Algol 60, particularly with regard to syntax [Wirth, 1982]. A detailed and well-judged reply to the critique by Habermann was written by Lecarme, who judged the merits and deficiencies on the basis of his experience with Pascal in both teaching and compiler design [Lecarme, 1975]. Another significant critique [Welsh, 1977] discusses the issue of structural vs. name equivalence of data types, a distinction that had unfortunately been left open in the definition of Pascal. It caused many debates until it was resolved by the Standards committee.

Perhaps the single most unfortunate construct was the variant record. It was provided for the purpose of constructing inhomogeneous data structures. Both for array and for a dynamic structures, in Pascal the element types must be fixed by type declarations. The variant record allows variations of the element types. The unfortunateness of the variant record of Pascal stems from the fact that it provides more flexibility than required to achieve this legitimate goal. In a dynamic structure, typically every element remains of the same type as defined by its creation. The variant record, however, allows more than the construction of heterogeneous structures, i.e. of structures with elements of different, although related types. It allows the type of elements themselves to change at any time. This additional flexibility has the regrettable property to require type checking at run-time for each access to such a variable or to one of its components. Most implementors of Pascal decided that this checking would be too expensive, enlarging code and deteriorating program efficiency. As a consequence, the variant record became a favourite feature to breach the type system by all programmers in love with tricks, which usually turn into pitfalls and calamities. Variant records also became a major hindrance to the notion of portabiliy of programs. Consider, for example, the declaration

```
VAR R:  RECORD maxspeed: INTEGER;
            CASE v: Vehicle OF
            truck: (nofwheels: INTEGER);
            vessel: (homeport: String)
        END
```

Here, the designator *R.nofwheels* is applicable only if *R.v* has the value *truck*, and *R.homeport* only if *R.v = vessel*. No compiler checks can safeguard against erroneous use of designators, which, in the case of assignment, may be disastrous, because the variant facility is used by implementations to save storage by overlaying the fields *nofwheels* and *homeport*.

With regard to input and output operations, Pascal separated the notions of data transfer (to or from an external medium) and of representation conversion (binary to decimal and vice-versa). External, legible media were to be represented as files (sequences) of characters. Representation conversion was expressed by special read and write statements that have the appearance of procedures but allowed a variable number of parameters. Whereas the latter was essentially a concession to programmers used to Fortran's I/O statements, the notion of sequence as a structural form was fundamental. Perhaps also in this case, providing sequences of any (even programmer-defined) element types was more than what was genuinely needed in practice. The consequence was that, in

contrast to all other data types, files require a certain amount of support from built-in run-time routines, mechanisms not explicitly visible from the program text. The successors of Pascal – Modula-2 and Oberon – later retreated from the notion of the file as a structural form at the same level as array and record. This became possible, because implementations of sequencing mechanisms could be provided through modules (library packages). In Pascal, however, the notion of modules was not yet present; Pascal programs were to be regarded as single, monolithic texts. This view may be acceptable for teaching purposes where exercises are reasonably compact, but it is not tenable for the construction of large systems. Nevertheless and surprisingly, Pascal compilers could be written as single Pascal programs.

Later Developments

As Pascal appeared to fulfil our expectations with regard to teaching, the compiler still failed to satisfy the stated goals with regard to efficiency in two aspects: First, the compiler as a relatively large stand-alone program resulted in fairly long "turn-around times" for students. In order to alleviate the problem, I designed a subset of the language containing those features that we believed were to be covered in introductory courses, and a compiler/interpreter package that fitted into a 16K-word block of store, which fell under the most-favoured program status of the computation center. The Pascal-S package was published in a report, and was one of the early comprehensive systems made widely available in source form [Wirth, 1981].

Comparable Fortran programs were still substantially faster, an undeniable argument in the hands of the Pascal adversaries. As we were of the opinion that structured programming supported by a structured language and efficiency of compilation and of produced code were not necessarily mutually exclusive, a project for a third compiler was launched, which on the one hand was to demonstrate the advantage of structured top-down design with step-wise refinement [Ammann, 1974], and on the other hand was to pay attention to generating high-quality code. This compiler was written by U. Ammann and achieved both goals quite admirably. It was completed in 1976 [Ammann, 1977].

Although the result was a sophisticated compiler of high quality and reliability, in hindsight we must honestly confess that the effort invested was incommensurate with its effect. It did not win over many engineers and even fewer physicists. The argument that Fortran programs "ran faster" was simply replaced by "our programs are written in Fortran". And what authorises us to teach structured, "better" programming to experts of ten years standing? Also, the code was generated for the CDC 6000 computer which –with its 60-bit word and super-RISC structure – was simply not well suited for the task. Much of Ammann's efforts went into implementing the attribute *packed* of records. Although semantically irrelevant, it was requested by considerations of storage economy on a computer with very long words. Having had the freedom to design not only the language but also the computer would have simplified the project considerably. In the event, the spread of Pascal came from another front.

Not long after the publication of the Pascal definition we received correspondence indicating interest in that language and requesting assistance in compiler construction, mainly from people who were not users of CDC computers. It was this stimulus which encouraged me to design a suitable computer architecture. A version of Ammann's compiler – easily derived from an early stage of the sequence of refinements – would generate code for this "ideal" architecture, which was described in the form of a Pascal program representing an interpreter. In 1973, the architecture became known as the P-machine, the code as P-code, and the compiler as the P-compiler. The P-kit consisted of the compiler in P-code and the interpreter as a Pascal source program [Nori, 1981]. Recipients could restrict their labour to coding the interpreter in their favourite assembler code, or proceed to modify the source of the P-compiler and replace its code generating routines. This P-system turned out to be the key to Pascal's spread onto many computers, but the reluctance of many to proceed beyond the interpretive scheme also gave rise to Pascal's classification as a "slow language", restricted to use in teaching.

Among the recipients of the P-kit was the team of K. Bowles at the University of California at San Diego (UCSD) around 1975. He had the foresight to see that a Pascal compiler for an interpretive system might well fit into the memory of a microcomputer, and he mustered the courage to try. Moreover, the idea of P-code made it easy to port Pascal on a whole family of micros and to provide a common basis on all of them for teaching. Microcomputers had just started to emerge, using early microprocessors like Intel's 8080, DEC's LSI-11, and Rockwell's 6502; in Europe, they were hardly known at the time.

Bowles not only ported our compiler. His team built an entire system around the compiler, including a program editor, a file system, and a debugger, thereby reducing the time needed for an edit-compile-test step dramatically over any other system in educational use. Starting in 1978, this UCSD-Pascal system spread Pascal very rapidly to a growing number of users [Bowles, 1980], [Clark, 1982]. It won more "Pascal friends" in a year than the systems used on large "main frames" had won in the previous decade. This phenomenal success had three sources: (1) A high-level language was available on microcomputers which would pervade educational institutions; (2) Pascal became supported by an integrated system instead of a "stand-alone" compiler; and (3), perhaps most importantly, Pascal was offered to a large number of computer novices, i.e. people who were not burdened by previous programming habits. In order to adopt Pascal, they did not have to give up a large previous investment in learning all the idiosyncracies of assembler or Fortran coding. The microcomputer made programming a public activity, hitherto exclusively reserved to the high-priests of computing centers, and Pascal effectively beat Fortran on microcomputers. By 1978, there existed over 80 distinct Pascal implementations on hosts ranging from the Intel 8080 microprocessor to the Cray-1 supercomputer. But Pascal's usefulness was not restricted to educational institutions; by 1980 all four major manufacturers of workstations (Three Rivers, HP, Apollo, Tektronix) were using Pascal for system programming.

Besides being the major agent for the spread of Pascal implementations, the P-system was significant in demonstrating how comprehensible, portable, and reliable a compiler and system program could be made. Many programmers learned much from the P-system, including implementors who did not base their work on the P-system, and others who had never before been able to study a compiler

in detail. The fact that a compiler was available in source form caused the P-system to become an influential vehicle of extracurricular education.

Several years earlier, attempts had been made to transport the Pascal compiler to other main-frame computers. In these projects no interpreter or intermediate code was used; instead they required the design of new generators of native code. The first of these projects was also the most successful. It was undertaken by J. Welsh and C. Quinn from the Queen's University at Belfast [Welsh, 1972]. The target was the ICL 1900 computer. The project deserves special mentioning, because it should be considered as one of the earliest, genuinely successful ventures that were later to be called software engineering efforts.

Since no CDC computer was available at Belfast, the goal was to employ a method that required as little work on a CDC machine as possible. What remained unavoidable would be performed during a short visit to ETH in Zürich. Welsh and Quinn modified the CDC-Pascal compiler, written in Pascal, by replacing all statements affecting code generation. In addition, they wrote a loader and an interpreter of the ICL-architecture, allowing some tests to be performed on the CDC-computer. All these components were programmed before the crucial visit, and were completed without any possibility of testing. In Zürich, the programs were compiled and a few minor errors were corrected within a week. Back in Belfast, the generated compiler code was executable directly by the ICL-machine after correction of a single remaining error.

This achievement was due to a very careful programming and checking effort, and it substantiated the claimed advantages to be gained by programming in a high-level language like Pascal, which provides full, static type checking. The feat was even more remarkable, because more than half of the week had to be spent on finding a way to read the programs brought from Belfast. Aware of the incompatibilities of character sets and tape formats of the two machines (7- vs. 9-track tapes), Welsh and Quinn decided to use punched cards as data carrier. Yet, the obstacles encountered were probably no less formidable. It turned out to be a tricky, if not downright impossible task, to read cards punched by an ICL machine with the CDC reader. Not only did the machines use different sets of characters and different encodings, but certain hole combinations were interpreted directly by the reader as end of records. The manufacturers had done their utmost best to ensure incompatibility! Apart from these perils, the travellers had failed to reckon with the thoroughness of the Swiss customs officers. The two boxes filled with some four thousand cards surely had to arouse their deep suspicion, particularly because these cards contained empty cubicles irregularly spaced by punched holes. Nevertheless, after assurances that these valuable possessions were to be re-exported anyway, the two might-be smugglers were allowed to proceed to perform their mission. Upon their return, the fact that now the holes were differently positioned luckily went unnoticed.

Other efforts to port the compiler followed; among them were those for the IBM 360 computers at Grenoble, the PDP-11 at Twente [Bron, 1976], and the PDP-10 at Hamburg [Grosse-Lindemann, 1976].

By 1973, Pascal had started to become more widely known and was being used in classrooms as well as for smaller software projects. An essential prerequisite for such acceptance was the availability of a user manual including tutorial material in addition to the language definition. Kathy Jensen

embarked on providing the tutorial part, and by 1973 the booklet was published by Springer-Verlag, first in their Lecture Notes Series, and after selling too fast, as an issue on its own [Jensen, 1974]. It was soon to be accompanied by a growing number of introductory textbooks from authors from many countries. The User Manual itself was later to be translated into many different languages, and it became a bestseller.

A dedicated group of Pascal fans was located at the University of Minnesota's computation center. Under the leadership and with the enthusiasm of Andy Mickel, a Pascal Users' Group (PUG) was formed, whose vehicle of communication was the Pascal Newsletter at first edited by G.H. Richmond (U. of Colorado) and later by Mickel. The first issue appeared in January 1974. It served as a bulletin board for new Pascal implementations, of new experiences and – of course – of ideas for improving and extending the language. Its most important contribution consisted in tracking all the emerging implementations. This helped both consumers to find compilers for their computers and implementors to coordinate their efforts.

At ETH Zurich, we had decided to move on towards other projects and to discontinue distribution of the compiler, and the Minnesota group was ready to take over its maintenance and distribution. Maintenance here refers to adaptation to continually changing operating system versions, as well as to the advent of the Pascal Standard.

Around 1977, a committee had been formed to define a standard. At the Southampton conference on Pascal, A. M. Addyman asked for help in forming a standards committee under the British Standards Institute (BSI). In 1978 representatives from industry met at a conference in San Diego hosted by K. Bowles to define a number of extensions to Pascal. This hastened the formation of a standards committee under the wings of IEEE and ANSI/X3. The formation of a Working Group within ISO followed in late 1979, and finally the IEEE and ANSI/X3 committees were merged into the single Joint Pascal Committee.

Significant conflicts arose between the US committee and the British and ISO committees, particularly over the issue of *conformant array parameters* (dynamic arrays). The latter became the major difference between the original Pascal and the one adopted by ISO, the other being the requirement of complete parameter specifications for parametric procedures and functions. The conflict on the issue of dynamic arrays eventually led to a difference between the standards adopted by ANSI on the one, and BSI and ISO on the other side. The unextended standard was adopted by IEEE in 1981 and by ANSI in 1982 [Cooper, 1983], [Ledgard, 1984]. The differing standard was published by BSI in 1982 and approved by ISO in 1983 [ISO, 1983], [Jensen, 1991].

In the meantime, several companies had implemented Pascal and added their own, supposedly indispensible extensions. The Standard was to bring them back under a single umbrella. If anything would have had a chance to make this dream come true, it would have been the speedy action of declaring the original language as the standard, perhaps with the addition of a few clarifications about obscure points. Instead, several members of the group had fallen prey to the devil's temptations: they extended the language with their own favourite features. Most of these features I had already contemplated in the original design, but dropped because either of difficulties in clear definition or efficient implementation, or because of questionable benefit to the programmer

[Wirth, 1975]. As a result, long debates started, requiring many meetings. When the committee finally submitted a document, the language had almost found its way back to the original Pascal. However, a decade had elapsed since publication of the report, during which individuals and companies had produced and distributed compilers; and they were not keen to modify them in order to comply with the late Standard, and even less keen to give up their own extensions. An implementation of the Standard was later published in [Welsh, 1986].

Even before publication of the Standard, however, a validation suite of programs was established and played a significant role in promoting compatibility across various implementations [Wichmann, 1983]. Its role even increased after the adoption of the Standard, and in the USA it made a Federal Information Processing Standard for Pascal possible.

The early 70s were the time when, in the aftermath of spectacular failures of large projects, terms like Structured Programming and Software Engineering were coined. They acted as symbols of hope for the drowning, and too often were believed to be panaceas for all the troubles of the past. This trend further raised interest in Pascal which – after all – was exhibiting a lucid structure and had been strongly influenced by E. W. Dijkstra's teachings on structured design. The 70s were also the years when in the same vein it was believed that formal development of correctness proofs for programs was the ultimate goal. C.A.R. Hoare had postulated axioms and rules of inference about programming notations (it later became known as Hoare-logic). He and I undertook the task of defining Pascal's semantics formally using this logic. However, we had to concede that a number of features had to be omitted from the formal definition (e.g. pointers) [Hoare, 1973].

Pascal therafter served as vehicle for the realization of program validators in at least two places, namely Stanford University and ETH Zürich. E. Marmier had augmented the compiler to accept assertions (in the form of marked comments) of relations among a program's variables holding after (or before) executable statements. The task of the assertion checker was to verify or refute the consistency of assertions and statements according to Hoare-logic [Marmier, 1975]. His was one of the earliest endeavours in this direction. Although it was able to establish correctness for various reasonably simple programs, its main contribution was to dispell the simple-minded belief that everything can be automated.

Pascal exerted a strong influence on the field of language design. It acted as a catalyst for new ideas and as a vehicle to experiment with them, and in this capacity gave rise to several successor languages. Perhaps the first was P. Brinch Hansen's *Concurrent Pascal* [Brinch Hansen, 1975]. It embedded the concepts of concurrent processes and synchronization primitives within the sequential language Pascal. A similar goal, but with emphasis on simulation of discrete event systems based on (quasi-) concurrent processes, led to *Pascal-Plus*, developed by J. Welsh and J. Elder at Belfast [Welsh, 1984]. A considerably larger language was the result of an ambitious project by Lampson et.al., whose goal was to cover all the needs of modern, large scale software engineering. Although deviating in many details and also in syntax, this language *Mesa* had Pascal as ancestor [Mitchell, 1978]. It added the revolutionary concept of modules with import and export relationships, i.e. of information hiding. Its compiler introduced the notion of separate – as distinct from independent – compilation of modules or packages. This idea was adopted later in *Modula-2*

[Wirth, 1982], a language that in contrast to Mesa retained the principles of simplicity, economy of concepts, and compactness which had led Pascal to success.

Another derivative of Pascal is the language *Euclid* [London, 1978]. The definition of its semantics is based on a formalism, just as the syntax of Algol 60 had been defined by the formalism BNF. Euclid carefully omits features that were not formally definable. *Object Pascal* is an extension of Pascal incorporating the notion of object-oriented programming, i.e. of the abstract data type binding data and operators together [Tesler, 1985]. And last but not least the language *Ada* [Barnes, 1980] must be mentioned. Its design was started in 1977 and was distinctly influenced by Pascal. It lacked, however, an economy of design without which definitions become cumbersome and implementations monstrous.

In Retrospect

I have been encouraged to state my assessment of the merits and weaknesses of Pascal, of the mistaken decisions in its design, and of its prospects and place in the future. I prefer not to do so explicitly, and instead to refer the reader to my own successive designs, the languages *Modula-2* [Wirth, 1982] and *Oberon* [Wirth, 1988]. Had I named them Pascal-2 and Pascal-3 instead, the questions might not have been asked, because the evolutionary line of these languages would have been evident.

It is also fruitless to question and debate early design decisions; better solutions are often quite obvious in hindsight. Perhaps the most important point was that someone did make decisions, in spite of uncertainties. Basically, the principle to include features that were well understood, in particular by implementors, and to *leave out* those that were still untried and unimplemented, proved to be the most successful single guideline. The second important principle was to publish the language definition *after* a complete implementation had been established. Publication of work done is always more valuable than publication of work planned.

Although Pascal had no support from industry, professional societies, or governmental agencies, it became widely used. The important reason for this success was that many people capable of recognizing its potential actively engaged themselves in its promotion. As crucial as the existence of good implementations is the availability of documentation. The conciseness of the original report made it attractive for many teachers to expand it into valuable textbooks. Innumerable books appeared in many languages between 1977 and 1985, effectively promoting Pascal to become the most widespread language used in introductory programming courses. Good course material and implementations are the indispensible prerequisites for such an evolution.

Pascal is still heavily used in teaching at the time of this writing. It may appear that it undergoes the same fate as Fortran, standing in the way of progress. A more benevolent view assigns Pascal the role of paving the way for successors.

Acknowledgements

I heartily thank the many contributors whose work played an indispensible role in making the Pascal effort a success, and who thereby directly or indirectly helped to advance the discipline of program design. Particular thanks go to C.A.R. Hoare for providing many enlightening ideas that flowed into Pascal's design, to U. Ammann, E. Marmier, and R. Schild for their valiant efforts to create an effective and robust compiler, to A. Mickel and his crew for their enthusiasm and untiring engagement in making Pascal widely known by establishing a User Group and a Newsletter, and to K. Bowles for recognizing that our Pascal compiler was also suitable for microcomputers and for acting on this insight. I also thank the innumerable authors of textbooks, without whose introductory texts Pascal could not have received the acceptance which it did.

References

[Ammann, 1974]
Ammann, U., The method of structured programming applied to the development of a compiler. *Int'l Computing Symposium 1973*, 93 – 99, North – Holland, 1974.

[Ammann, 1977]
Ammann, U., On Code Generation in a Pascal Compiler.
Software – Practice and Experience, 7, 391–423 (1977).

[Barnes, 1980]
An Overwiew of Ada. *Software – Practice and Experience, 10*, 851 – 887 (1980).

[Bowles, 1980]
Bowles, K. L. *Problem solving using Pascal.* Springer–Verlag, 1977.

[Brinch Hansen, 1975]
Brinch Hansen, P., The Programming Language Concurrent Pascal,
IEEE Trans. Software Eng. 1, 2, 199 – 207, (1975).

[Bron, 1976]
Bron, C., and W. de Vries. A Pascal Compiler for the PDP–11 Minicomputers.
Software – Practice and Experience, 6, 109–116 (1976).

[Clark, 1982]
Clark, R., and S. Koehler, *The UCSD Pascal Handbook.* Prentice–Hall, 1982.

[Cooper, 1983]
Cooper, D., *Standard Pascal, User Reference Manual.* Norton, 1983.

[Dijkstra, 1966]
Dijkstra, E. W., Structured Programming, Tech. Report, Univ. of Eindhoven, 1966.
also in: Dahl, O. – J. et al, *Structured Programming*, London: Academic Press 1972.

[Grosse-Lindmann, 1976]
Grosse-Lindemann, C. O., and H. H. Nagel, Postlude to a Pascal-Compiler Bootstrap on a DECSystem-10, *Software – Practice and Experience, 6*, 29–42, (1976).

[Habermann, 1973]
Habermann, A. N., Critical comments on the programming language Pascal, *Acta Informatica 3*, 47 – 57 (1973).

[Hoare, 1972]
Hoare, C.A.R., Notes on Data Structuring.
In: Dahl, O. – J. et al, *Structured Programming*, London: Academic Press 1972.

[Hoare, 1973]
Hoare, C.A.R. and N. Wirth, An axiomatic definition of the programming language Pascal, *Acta Informatica 2*, 335 – 355 (1973)

[Hoare, 1980]
Hoare, C.A.R., The Emperor's Old Clothes. *Comm. ACM, 24*, 2 (Feb. 1980), 75 – 83 (Feb. 1980).

[ISO, 1983]
International Organization for Standardization, *Sepcification for Computer Programming Language Pascal*, ISO 7185–1982.

[Jensen, 1974]
Jensen, K. and N. Wirth, *Pascal – User Manual and Report*, Springer-Verlag, 1974.

[Jensen, 1991]
Jensen, K., and N. Wirth. Revised by A. B. Mickel and J. F. Miner, *Pascal, User Manual and Report, ISO Pascal Standard*. Springer-Verlag, 1991.

[Lecarme, 1975]
Lecarme, O. and P. Desjardins, More comments on the programming language Pascal, *Acta Informatica 4*, 231 – 243 (1975)

[Ledgard, 1984]
Ledgard, H. *The American Pascal Standard*. Springer-Verlag, 1984.

[London, 1978]
London, R.L., J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof Rules for the Programming Language Euclid. *Acta Informatica 10*, 1 – 26 (1978).

[Marmier, 1975]
E. Marmier. Automatic Verification of Pascal Programs.
ETH-Dissertation No. 5629, Zürich, 1975

[Mitchell, 1978]
Mitchell, J. G., W. Maybury, R. Sweet. Mesa Language Manual.
Xerox PARC Report CSL-78-1 (1978).

[Naur, 1963]
Naur, P., (ed.) Revised report on the algorithmic language Algol 60,
Comm. ACM 3, 299 – 316 (1960), and *Comm. ACM 6*, 1 – 17 (1963)

[Nori, 1981]
Nori, K.V., et al., The Pascal P–code compiler: Implementation notes.
In: D.W. Barron, ed., *Pascal – The language and its implementation*. Wiley 1981.

[Tesler, 1985]
Tesler, L., Object Pascal Report.
Structured Programming (formerly *Structured Language World*), *9*, 3, (1985), 10 – 14.

[van Wijngaarden, 1969]
van Wijngaarden, A., (Ed.), Report on the algorithmic language Algol 68.
Numer. Math. 14, 79 – 218 (1969)

[Welsh, 1972]
Welsh, J., and C. Quinn. A Pascal Compiler for ICL 1900 Series Computers.
Software, Practice and Experience 2, 73–77 (1972).

[Welsh, 1977]
Welsh, J., W. J. Sneeringer, and C.A.R. Hoare, Ambiguities and Insecurities in Pascal.
Software, Practice and Experience, *7*, (1977), 685 – 696.
Also in: D.W. Barron, ed., *Pascal – The language and its implementation*. Wiley 1981.

[Welsh, 1984]
Welsh, J., and D. Bustard, *Sequential Program Structures*, Prentice–Hall Int'l, 1984.

[Welsh, 1986]
Welsh, J., and A. Hay, *A Model Implementation of Standard Pascal*. Prentice–Hall Int'l, 1986.

[Wichmann, 1983]
Wichmann, B., and Ciechanowicz, *Pascal Compiler Validation*. Wiley, 1983.

[Wirth, 1966]
Wirth, N. and C.A.R. Hoare, A Contribution to the development of ALGOL,
Comm. ACM 9, 6, 413 – 432 (June 1966)

[Wirth, 1970]
Wirth, N., The Programming Language Pascal,
Tech. Rep. 1, Fachgruppe Computer–Wissenschaften, ETH, Nov. 1970, and
Acta Informatica 1, 35 – 63 (1971)

[Wirth, 1971a]
Wirth, N., Program Development by Step–wise Refinement.
Comm. ACM 14, 4, 221–227 (April 1971)

[Wirth, 1971b]
Wirth, N., The design of a Pascal compiler.
Software, Practice and Experience 1, 309 – 333 (1971).

[Wirth, 1975]
Wirth, N. An assessment of the programming language Pascal.
IEEE Trans. on Software Eng., 1, 2 (June 1975), 192 – 198.

[Wirth, 1981]
Wirth, N. Pascal–S: A subset and its implementation.
In: D.W. Barron, ed., *Pascal – The language and its implementation*. Wiley 1981.

[Wirth, 1982]
Wirth, N., *Programming in Modula–2*. Springer–Verlag, 1982.

[Wirth, 1988]
Wirth, N., The Programming Language Oberon.
Software, Practice and Experience 18, 7 (July 1988), 671 – 690.