III

Pascal Session

Chairman: *Jacques Cohen*
Discussant: *Andrew B. Mikel*

RECOLLECTIONS ABOUT THE DEVELOPMENT OF PASCAL

N. Wirth

Institut für Computersysteme, ETH Zurich
CH-8092 Zurich

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 summarizes 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

3.1 EARLY HISTORY

The programming language Pascal was designed in the years 1968–1969, 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 to make it suitable for purposes other than numerical computation. To 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 and unafraid of constructing features that were largely untried and whose consequences for implementors remained a matter of speculation, who were 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 favored 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 in 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 1970s, 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 from the U.S. 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, and partly because it lacked adequately flexible data structuring facilities. I therefore decided to pursue my original goal of designing 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. 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 Zurich (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 helping 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].

3.2 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, that is, 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 afterthoughts. 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 recognized. In many instances, a decision had to be taken whether to solve a case in a clear-cut fashion, or to remain compatible with ALGOL. These options sometimes were mutually exclusive. In retrospect, the decisions in favor 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, nonterminating 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: 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 refer here 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 parameter-

ized 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 recognized 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 could only be accessed directly, record structured variables could only be accessed via references (pointers), that is, 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, that is, 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, that is, 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 easy 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 requires 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 or her 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 Hoare's *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 records with files, to name just a few possibilities. Naturally, implementations would have to impose certain limits as 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 such as conditional expressions, the exponentiation operator, and local blocks, which were all present in ALGOL 60, he reproached Pascal for retaining the much-cursed **goto** 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 versus 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 nonhomogeneous data structures. Both for array and for dynamic structures in Pascal, the element types must be fixed by type declarations. The variant record allows variations of the element types. The unfortunate aspect 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, that is, 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 of requiring 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 favorite 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 portability 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.

3.3 LATER DEVELOPMENTS

Even though Pascal appeared to fulfill our expectations in 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-favored 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 not commensurate 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 authorizes 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 packet 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 any 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 that 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 labor to coding the interpreter in their favorite 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 to 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 such as 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 "mainframes" had won in the previous decade. This phenomenal success had three sources: (1) a high-level language, which would pervade educational institutions, was available on microcomputers; (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, that is, 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 a great deal 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 mainframe 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 Queen's University, Belfast [Welsh 1972]. The target was the ICL 1900 computer. The project deserves special mention, because it should be considered as one of the earliest, genuinely successful ventures that were later to be called software engineering efforts.

As 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 Zurich. 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 Zurich, 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 such as 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 (seven- versus nine-track

tapes), Welsh and Quinn decided to use punched cards as the 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 travelers 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 reexported 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 very quickly, 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 computer 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 (University of Colorado) and later by Mickel. The first issue appeared in January 1974. It served as a bulletin board for new Pascal implementations, for new experiences and—of course—for ideas of 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 U.S. 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 one hand, and BSI and ISO on the other. 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 indispensable extensions. The standard was to bring them back under a single umbrella. If anything might 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 favorite features. Most of these features I had already contemplated in the original design, but dropped either because 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 United States it made a Federal Information Processing Standard for Pascal possible.

The early 1970s were the time when, in the aftermath of spectacular failures of large projects, terms such as 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 1970s 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 thereafter served as a vehicle for the realization of program validators in at least two places—namely, Stanford University and ETH Zurich. 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 endeavors in this direction. Although it was able to establish correctness for various reasonably simple programs, its main contribution was to dispel 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 its ancestor [Mitchell 1978]. It added the revolutionary concept of modules with import and export relationships, that is, 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 that 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, that is, 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 became cumbersome and implementations monstrous.

3.4 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 government 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 implementation 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 indispensable prerequisites for such an evolution.

Pascal is still widely 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.

ACKNOWLEDGMENTS

I heartily thank the many contributors whose work played an indispensable 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 that it did.

REFERENCES

[Ammann, 1974] Ammann, U., The Method of Structured Programming Applied to the Development of a Compiler, in *International Computing Symposium 1973*, Amsterdam: North-Holland, 1974, pp. 93–99.

[Ammann, 1977] Ammann, U., On Code Generation in a Pascal Compiler, *Software—Practice and Experience*, Vol. 7, 1977, pp. 391–423.

[Barnes, 1980] Barnes, , An Overwiew of Ada, *Software—Practice and Experience*, Vol. 10, 1980, pp. 851–887.

[Bowles, 1980] Bowles, K. L., *Problem Solving Using Pascal*. Springer-Verlag, 1977.

[Brinch Hansen, 1975] Brinch Hansen, P., The Programming Language Concurrent Pascal, *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, 1975, pp. 199–207.

[Bron, 1976] Bron, C., and W. de Vries, A Pascal Compiler for the PDP-11 Minicomputers, *Software—Practice and Experience*, Vol. 6, 1976, pp. 109–116.

[Clark, 1982] Clark, R., and S. Koehler, *The UCSD Pascal Handbook*, Englewood Cliffs, NJ: Prentice-Hall, 1982.

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

[Dijkstra, 1966] Dijkstra, E. W., *Structured Programming*, Technical Report, University of Eindhoven, 1966. Also in Dahl, O.-J. *et al*,. *Structured Programming*, London: Academic Press, 1972.

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

[Habermann, 1973] Habermann, A. N., Critical Comments on the Programming Language Pascal, *Acta Informatica* Vol. 3, 1973, pp. 47–57.

[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*, Vol. 2, 1973, pp. 335–355.

[Hoare, 1980] Hoare, C. A. R., The Emperor's Old Clothes. *Communications of the ACM*, Vol. 24, No. 2 , Feb. 1980, pp. 75–83.

[ISO, 1983] International Organization for Standardization, *Specification 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*, Vol. 4, 1975, pp. 231–243.

[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*, Vol. 10, 1978, pp. 1–26.

[Marmier, 1975] Marmier, E., *Automatic Verification of Pascal Programs*, ETH Dissertation No. 5629, Zurich, 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, *Communications of the ACM*, Vol. 3, 1960, pp. 299–316; *Communications of the ACM*, Vol. 6, 1963, pp. 1–17.

[Nori, 1981] Nori, K.V. *et al.*, The Pascal P-code Compiler: Implementation Notes, in *Pascal—The Language and Its Implementation*. D.W. Barron, ed., New York: John Wiley & Sons, 1981.

[Tesler, 1985] Tesler, L., Object Pascal Report,. *Structured Programming* (formerly *Structured Language World*), Vol. 9, No. 3, 1985, pp. 10–14.

[van Wijngaarden, 1969] van Wijngaarden, A., Ed., Report on the Algorithmic Language ALGOL 68, *Numer. Math.* Vol. 14, 1969, pp. 79–218.

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

[Welsh, 1977] Welsh, J., W. J. Sneeringer, and C. A. R. Hoare, Ambiguities and Insecurities in Pascal, *Software—Practice and Experience*, Vol. 7, 1977, pp. 685–696. Also in D. W. Barron, Ed., *Pascal—The Language and its Implementation*, New York: John Wiley & Sons, 1981.

[Welsh, 1984] Welsh, J., and D. Bustard, *Sequential Program Structures*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
[Welsh, 1986] Welsh, J., and A. Hay, *A Model Implementation of Standard Pascal*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
[Wichmann, 1983] Wichmann, B., and Ciechanowicz, *Pascal Compiler Validation*, New York: John Wiley & Sons, 1983.
[Wirth, 1966] Wirth, N. and C. A. R. Hoare, A Contribution to the Development of ALGOL, *Communications of the ACM*, Vol. 9, No. 6, June 1966, pp. 413–432.
[Wirth, 1970] Wirth, N., *The Programming Language Pascal*, Techical Report 1, Fachgruppe Computer-Wissenschaften, ETH, Nov. 1970; *Acta Informatica*, Vol. 1,1971, pp. 35–63.
[Wirth, 1971a] Wirth, N., Program Development by Step-wise Refinement, *Commnications of the ACM*, Vol. 14, No. 4, Apr. 1977, pp. 221–227.
[Wirth, 1971b] Wirth, N., The Design of a Pascal Compiler, *Software—Practice and Experience*, Vol. 1, 1971, pp. 309–333.
[Wirth, 1975] Wirth, N. An assessment of the programming language Pascal. *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, June 1975, pp. 192–198.
[Wirth, 1981] Wirth, N., Pascal-S: A Subset and its Implementation, in *Pascal—The Language and its Implementation*, D. W. Barron, Ed., New York: John Wiley & Sons, 1981.
[Wirth, 1982] Wirth, N., *Programming in Modula-2*, Springer-Verlag, 1982.
[Wirth, 1988] Wirth, N., The Programming Language Oberon, *Software—Practice and Experience*, Vol. 18, No. 7, July 1988, pp. 671–690.

TRANSCRIPT OF PRESENTATION

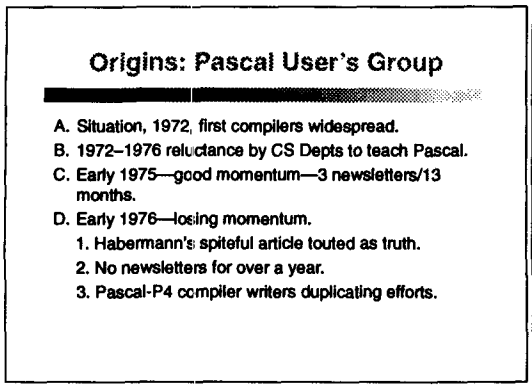
Editor's note: Niklaus Wirth's presentation closely followed his paper and we have omitted it with his permission.

TRANSCRIPT OF DISSCUSSANT'S REMARKS

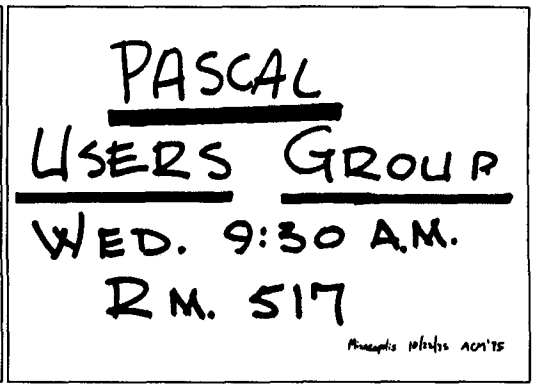
SESSION CHAIR JACQUES COHEN: The discussant is Andy Mickel, who is the Computer Center Director at the Minneapolis College of Art and Design. Prior to his current position, he worked for ten years at the University of Minnesota Computer Center, where he managed the language processor's group supporting micros through CDC and VAX mainframes, up to the Cray-1. He cofounded the Pascal Users Group at ACM '75, and assumed editorship of the *Pascal Newsletter* in the summer of 1976. Through 1979, he coordinated the Users Group and edited the quarterly, *Pascal News*. Beginning with the Southampton Pascal Conference in 1977, he facilitated the international effort to standardize and promote the use of Pascal. After the standard was complete in 1983, he and his colleague, Jim Miner, revised the popular tutorial, *Pascal User Manual and Report*, in a third edition to conform to the standard. From 1983 to 1989, he worked at Apollo Computer, where in 1986 he led the project to develop the first commercial, two-pass Modula-2 compiler.

ANDREW B. MIKEL: My talk is on the origins of the Pascal Users Group and its effect on the use and dissemination of Pascal. I was introduced to programming languages by Daniel Friedman (who is here today), in my senior year at the University of Texas at Austin, 1971. He taught me language definition and programming language semantics. I went on to grad school at the University of Minnesota, where I got a job at the Computer Center and became interested in promoting the use of good, practical, general-purpose programming languages, but we had a Control Data machine. And that was a good thing, but along came Pascal and my efforts to promote it were met with resistance from FORTRAN users and other skeptics. So, my goal became to make the world safe for a language, that I, a nonguru, mortal programmer enjoyed using.

(SLIDE 1) I have a slide here where I have listed the sort of situation we found ourselves in 1970. Even though there were some Pascal compilers coming into use, there was a reluctance, particularly by American computer science departments, to teach Pascal. Along came George Richmond, from the University of Colorado, who started the *Pascal Newsletter* and created some momentum in our



SLIDE 1

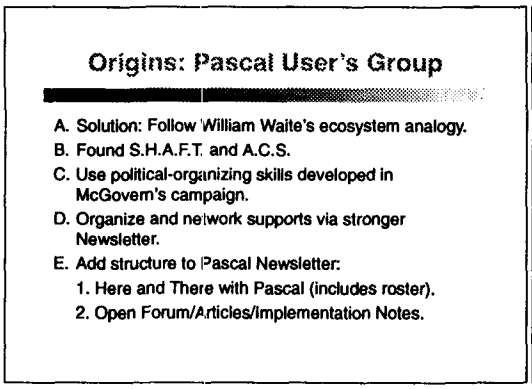


SLIDE 2

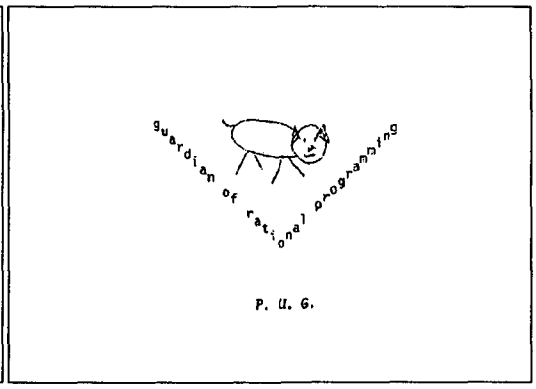
eyes by producing three newsletters in 13 months. But, then the newsletters stopped and people were quoting Habermann's article as if they had read it and had not studied Pascal. A lot of people using Pascal-P4 to build compilers were duplicating each others' efforts.

(SLIDE 2) We have a photocopy of the original sign used by Richard Cichelli at ACM '75 in Minneapolis, announcing a Pascal Users Group. I had never heard of this before. But I went. Bob Johnson, his friend from Saint Cloud State (Richard was from Lehigh University), Charles Fischer and Richard LeBlanc (who is here) from the University of Wisconsin, and 31 other people showed up in a hotel room. We talked about what to do. A lot of us were pretty angry.

(SLIDE 3) We formed a strategy. We decided to follow William Waite's advice in the *Software—Practice and Experience* article editorial about programming languages as organisms in an ecosystem. And if FORTRAN was a tough weed, we needed to grow a successful competing organism and starve FORTRAN at the roots. So we talked about forming little clubs like Society to Help Abolish FORTRAN Teaching (that's SHAFT), or the American COBOL Society "dedicated to the elimination of COBOL in our lifetime" (like the American Cancer Society). I was fresh from working at George McGovern's political campaign and you have to remember that the early seventies was the era of Watergate and the truth, integrity and full disclosure. So, in 1976, I decided to pro-actively mail directly to every college with a math department a notice that we had a Pascal Users Group. We rapidly built a large mailing list. We used the newsletter as a means to organize this whole activity. It was



SLIDE 3



SLIDE 4

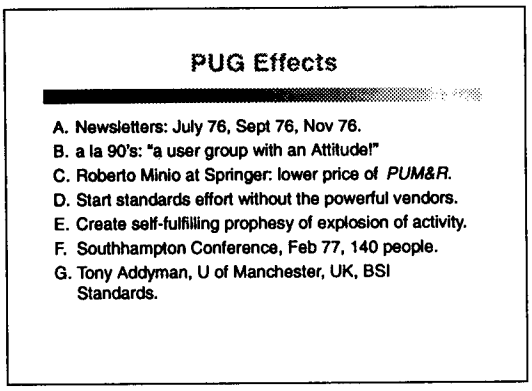
much like the language supplements to *SIGPLAN Notices*. The average size was 150 pages.

(SLIDE 4) We didn't take ourselves too seriously. Here is what the guardian of rational programming, the PUG mascot looked like. It is sort of a weakling looking dog, a pug dog.

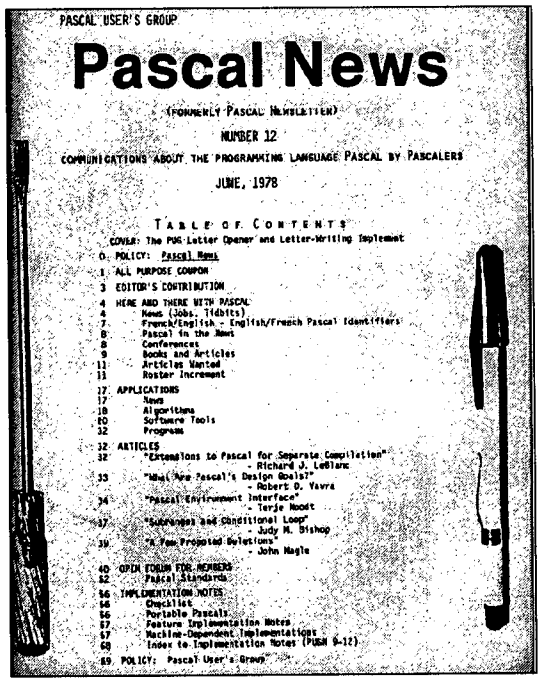
(SLIDE 5) We started getting things back on track. We had an aggressive attitude. In the 90s we would say we were a User Group with "an attitude." We indulged in self-fulfilling prophesies. We tried to make it look like there was a lot going on around Pascal by publishing everything that we could find on it. And lo and behold, there was a lot going on with Pascal because some of the neutral bystanders became less afraid and started joining in. David Barron in Southampton scheduled the first Pascal Conference. One hundred and forty people attended. At that conference, Tony Addyman teased people with the idea of starting a formal standards effort. That was the beginning of standards, which took place in Europe, something a little different from what most American computer manufacturers were used to. Some of the early influences at that time were Wilhelm Burger at the University of Texas at Austin; John Strait, Jim Miner, and John Easton at Minnesota; Judy Bishop with David Barron at Southampton; Arthur H. J. Sale at the University of Tasmania in Australia; and Helmet Weber at the Bavarian University of Munich.

(SLIDE 6) This is an sample cover of *Pascal News*, which sort of illustrates its use of a self-organizing tool. The newsletters contain information about users, compilers, standards, vendors, applications, teaching, textbooks, and even the roster of all the members. My plan was if my Computer Center jerked the support out from under me, anybody could pick up the User Group and keep going.

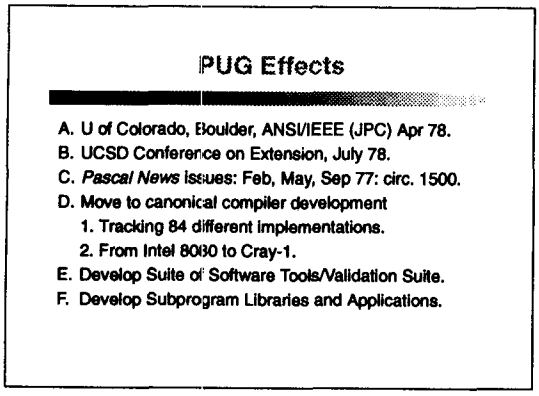
(SLIDE 7) So, we were finally on a roll. The American committee starts, we get them cooperating without trying to dominate the standards process—we argued that it was a European Language being standardized by Europeans. We derailed the UCSD conference and "extensions-with-Pascal" people. The people at UCSD (the



SLIDE 5



SLIDE 6



SLIDE 7

SLIDE 8

bad guys) were confusing the language definition with its implementation. Circulation of *Pascal News* increased. We also sorted out a lot of duplicated efforts among the PDP-11 and IBM 370 implementations: There were 20 different compiler efforts. That is not counted in the total of 84; that is just the ones that weren't canonical. There was a lot of software written in Pascal becoming available.

(SLIDE 8) Then all hell broke loose. This article appeared in *ComputerWorld* in April 1978 and down here it says, "You can join the two thousand members of the Pascal Underground by sending four dollars to the Pascal Users Group." My mail went from 15 pieces a day to 80-something pieces a day.

(SLIDE 9) This is a picture of me up against a wall pasted with copies of all the articles on Pascal out of the industry journals.

(SLIDE 10) This is sample of what kind of coverage we were starting to get in the industry press in 1978: *ComputerWorld, Byte, Electronics Magazine, IEEE Computer, and Datamation.* By the next year, our circulation was up to 4,600 in 43 countries.

(SLIDE 11) So we were riding high. Arthur Sale produced a little cartoon, "Pascal Riding on the Microwave."

(SLIDE 12) From an American point of view, industry led over academia, and here are some of the projects. A lot of them were internal to the companies over here that nobody really knew about. It was only in 1981, that the commercial versions of UCSD Pascal started to appear. I want to point out that Three Rivers PERQ, Tektronix, and the Hewlett-Packard 9000 are workstations.

(SLIDE 13) This may be controversial, too. It may be fortuitous that the first compiler was on Control Data equipment because at that time in the seventies, CDC had a disproportionately large installed base at universities. For example, the size of the student body at Minnesota was 55,000, at UT Austin it was 40,000, 45,000 at UC Berkeley, and all had Control Data equip-

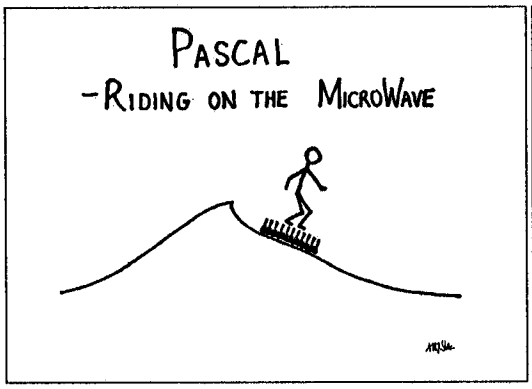
SLIDE 9

PUG Effects – Press

A. Industry Press articles.
   1. *ComputerWorld*, page 24, 1 page, April, 1978.
   2. *Byte*: cover story, August, 1978.
   3. *Electronics*: cover mention, October, 1978.
   4. *IEEE Computer*: cover mention, April, 1979.
   5. *Datamation*: cover mention, July 1979.

B. *Pascal News*: Jan, Sep, Oct 79: circ 4600 in 43 countries.

SLIDE 10



SLIDE 11

PUG Effects – Industry over Academe

A. Control Data 1700, 1975.
B. Texas Instruments, ASC, 990 and 9900, 1976.
C. Control Data/NCR Joint Project, 1979.
D. Three Rivers PERQ, 1978.
E. Tektronix, 1978.
F. Apollo DOMAIN, 1980.
G. Apple Apple][, 1981.
H. Hewlett Packard 9000, 1981.

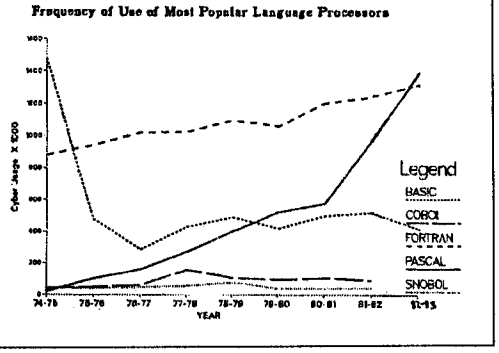
SLIDE 12

PUG Effects – Industry over Academe

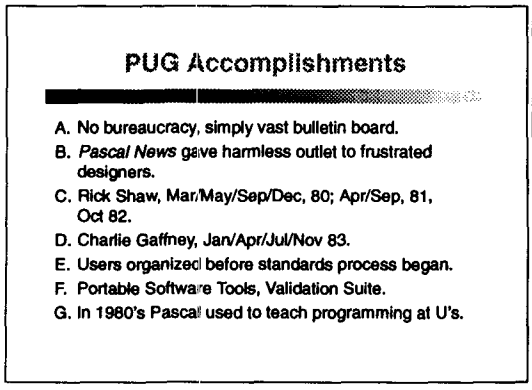
A. CDC 6000/7000/Cyber sites at large universities in USA.
   1. U of Minnesota, 55,000.
   2. U of Texas at Austin, 40,000.
   3. U of California, Berkeley, 35,000.

B. Last to teach Pascal.
   1. M.I.T.
   2. Stanford U.
   3. Carnegie-Mellon U.

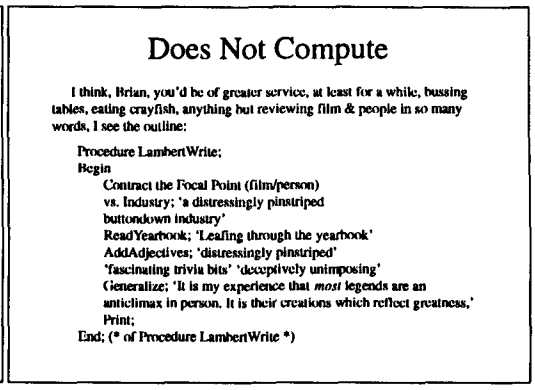
SLIDE 13



SLIDE 14



SLIDE 15



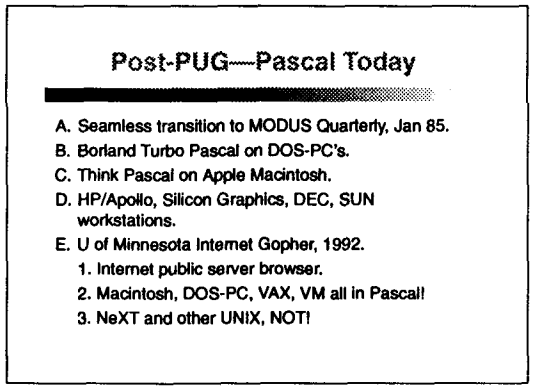
SLIDE 16

ment exceeding those small, more elite schools. I was chagrined that the last universities in the United States to use Pascal as a teaching language were MIT, Stanford, and Carnegie Mellon.

(SLIDE 14) This is an example of the usage at the University of Minnesota and of the number of compilations by language from 1974 to 1983. If we just take 1975 as a baseline, there were 900,000 FORTRAN compilations to 100,000 to Pascal. And by 1983, it was 1.3 million FORTRAN to 1.4 million Pascal—that is, not including all the compiles that were done on TERAK (which were LSI-11's and Apple II's running UCSD Pascal).

(SLIDE 15) So, what were our accomplishments? This is sort of a summary. The Users Group was not a bureaucracy. We could fold it up at any time. It was simply a vast bulletin board. I hope I can reassure Niklaus that the *Pascal News* articles provided a harmless outlet to frustrated language designers who wanted to extend Pascal. It dissipated their energy harmlessly. My successors were Rick Shaw (not a transportation vehicle) and Charlie Gaffney. One of the important things about the standards process is that the users were organized before the standards process began.

(SLIDE 16) This is a letter to the editor of a mainstream newspaper in the Twin Cities that is in the form of a Pascal program—so that by 1983, Pascal had even leaked out into the common culture. It's a critique of a movie or film critic.



SLIDE 17

(SLIDE 17) It turns out that *Pascal News* may have stopped publishing in 84 but there was a seamless transition to *MODUS Quarterly*, the Modula-2 Users Society. Here is the list of the current Pascal compilers, and the most famous application newly written in Pascal: the University of Minnesota Internet Gopher. And the versions for clients for Macintosh, DOS PC, VAX, and IBM VM are all in Pascal, but none of the UNIX boxes.

TRANSCRIPT OF QUESTION AND ANSWER SESSION

ANDREW BLACK (Digital Cambridge Research Laboratory): The omission of dynamic arrays from Pascal was as much a mistake as the omission of recursion. True or False?

WIRTH: In hindsight, true. You have to recall that I had not only to make language design decisions but also to implement them. Under these circumstances it is a question of whether to include more features, or to get ahead first with those needed to implement the compiler. In principle, you are right: Dynamic arrays should have been included. (Note: Recursion was excluded in a preliminary version only.)

HERBERT KLAEREN (University of Tubingen): The current ISO draft of Modula-2 contains a full formal semantics. Do you consider this useful?

WIRTH: Not really. It makes the report too voluminous for the reader. The Modula-2 report is 45 pages and the formal standardization document is about one thousand pages long. The definition of the empty statement alone takes a full page. The difference is intolerable.

MARTIN CAMPBELL-KELLY (University of Warwick): If, on a scale of excellence in programming language design, FORTRAN was awarded two, and Pascal was awarded five, what score would Modula-2 deserve?

WIRTH: Six. (In the Swiss school system, one is the worst and six is the best grade.)

BRENT HAILPERN (IBM Research): Did you foresee the type "abuse" resulting from variant records? If not, what crossed your mind when the loophole was discovered?

WIRTH: I did not foresee it, but I was told so by Tony Hoare. The reason for including the variant record was simply the need for it, and the lack of knowledge of how to do better. In the compiler itself several variants of records occur in the symbol table. For reasons of storage economy, variants are needed in order to avoid unused record fields. With memory having become available in abundance, the facility is now less important. Furthermore, better, type-safe constructs are known, such as type extensions (in Oberon).

RANDY HUDSON (Infometrics): What relationship do you see between the design of Pascal and the design of Ada?

WIRTH: I believe Ada did inherit a few ideas from Pascal, such as structures, concepts, and perhaps philosophy. I think the main differences stem from the fact that I, as an individual, could pick the prime areas of application, namely teaching and the design of systems of moderate complexity. Ada was devised according to a large document of requirements that the designers could not control. These requirements were difficult, extensive, partly inconsistent, and some were even contradictory. As a result, its complexity grew beyond what I considered acceptable.

BOB ROSIN: Was the P-code concept inspired by earlier work? For example, the Snobol-4 implementation?

WIRTH: It wasn't particularly the Snobol-4 implementation. The technique of interpreters was well known at the time.

ALLEN KAY (Apple Computers): Your thesis and Euler were going in a pretty interesting direction. Why did you go back to more static languages?

WIRTH: For a thesis topic you have to do something that is new and original, you are expected to explore new ways. Afterwards, as an engineer, I felt the need for creating something that is pragmatic and useful.

ROSS HAMILTON (University Of Warwick): Why did Pascal not allow source code to be split across multiple files (as in FORTRAN II)?

WIRTH: We didn't perceive a need to provide such a facility. It is an implementation consideration, but not one of language design.

ELLEN SPERTUS (MIT): You discussed the IEEE/ANSI standard for Pascal. There is also the *de facto* Borland Turbo Pascal definition. Could you comment on the differences, in effect and content, of standards from democratic committees and from single companies?

WIRTH: I think that would take too much time here. But the point is well taken that the actual standard for Pascal has been defined by Borland, just by producing a compiler and distributing it cheaply and widely. If you talk to anyone about Pascal today, probably Turbol Pascal from Borland is meant.

Borland extended Pascal over the years, and some of these extensions were not as well integrated into the language as I would have wished. I was less compromising with my extensions and therefore gave the new design a new name. For example, what Borland calls **Units** we call **Modules**. However, **Units** are inclusions in the form of source code, whereas **Modules** are separate units of compilation, fully type checked and linked at load time.

JEAN SAMMET (Programming Language Consultant): You mentioned COBOL twice—once in connection with control structures and once in connection with records. Please clarify whether COBOL had any influence on Pascal and if so, in what way?

WIRTH: Pascal inherited from COBOL the record structure, in fact indirectly via Tony Hoare's proposal included in Pascal's predecessor, ALGOL W.

DANIEL J. SALOMON (University of Manitoba): Although the pragmatic members of the ALGOL 68 Working Group have been shown, by time, to be correct, has not the ALGOL 68 report been valuable, flaws and all?

WIRTH: It has fostered a method of defining languages with precision and without reference to implementations. Its failure was probably due to driving certain ideas, such as "orthogonality" (mutual independence of properties) to extremes. The design was too dogmatic to be practical.

STAVROS MACRAKIS (OSF Research Institute): The original Pascal was small, limited in function, and completely defined. ISO's Pascal is large, comprehensive, and presumably well defined. What happened? Is it fair to consider ISO Pascal a diagnosis of Pascal's failings?

WIRTH: I rather consider the standard as the belated result of a long-winded effort to make the definition more precise, and at the same time, of extending the language.

HERBERT KLAEREN (University of Tubingen): I always wondered why Pascal's dynamic data structures did not turn recursive data structures into "first class citizens." Instead of having the programmer himself handle explicit pointers (and doing all kinds of mischief with them), the compiler could have detected the need of introducing implicit pointers automatically (and transparently). Did this idea ever come up and if so, why did you reject it?

WIRTH: We did not see a way to incorporate recursive structures in a way that was as flexible and as effective as explicit pointers. I could not wait until a solution appeared; time was precious and the need for an implementation language pressing.

HERBERT KLAEREN (University of Tubingen): Given your comments on the Pascal standardization, what are your feelings about the ongoing ISO standardization of Modula-2? Wouldn't it be a good idea if the creator of a programming language gave, along with its definition, a list of the features contemplated, but dropped, along with the reasons for dropping them?

WIRTH: I am sure this would be a good idea, preventing later designers from having to repeat the same considerations.

ADAM RIFKIN (California Institute of Technology): Why do you think the minority report signed in Munich (December 1968) had so many followers? Can you cite one main reason?

WIRTH: The Working Group 2.1 had been divided since the advent of A. van Wijngaarden's initial draft design in late 1965. The decision to pursue his draft rather than mine was taken in Warsaw in late 1966. The principal difference was that I had followed the goal of creating a successor to ALGOL 60 by essentially building upon ALGOL 60 and widening its range of application by adding data structures. Van Wijngaarden wanted to create an entirely new design, in fact another milestone (not millstone) in language design. The faction of "pragmatists" foresaw that it would take a long time to implement van Wijngaarden's proposal, and that by the time of its completion the majority of ALGOL users would have chosen another language already. (Note that ALGOL 68 compilers did not become available before 1972.)

Question is to ANDY MICKEL: The speaker mentioned an early anti-Pascal letter. Could you say more about that?

MIKEL: That was an article that appeared in *Acta Informatica* by A. N. Habermann, who wrote the paper called "Critical Comments on the Programming Language in Pascal" that Niklaus Wirth cited in his talk. It was countered by a reply from Oliver Lacarme and Pierre Dejardins. A lot of the self appointed experts at the University of Minnesota Computer Science Department preferred to beat us up in Computer Center about how Pascal should not have been taught, based on Habermann's paper, rather than even trying to use Pascal and see for themselves.

BIOGRAPHY OF NIKLAUS WIRTH

Niklaus Wirth received the degree of Electronics Engineer from the Swiss Federal Institute of Technology (ETH) in Zurich in 1958. Thereafter he studied at Laval University in Quebec, Canada, and received the M.Sc. degree in 1960. At the University of California at Berkeley he pursued his studies, leading to the Ph.D. degree in 1963. Until 1967, he was Assistant Professor at the newly created Computer Science Department at Stanford University, where he designed the programming languages PL360 and—in conjunction with the IFIP Working Group 2.1—ALGOL W. In 1967, he

became Assistant Professor at the University of Zurich, and in 1968 he joined ETH Zurich, where he developed the languages Pascal between 1968 and 1970 and Modula-2 between 1979 and 1981.

Further projects include the design and development of the Personal Computer Lilith, a high-performance workstation, in conjunction with the programming language Modula-2 (1978–1982), and the 32-bit workstation computer Ceres (1984–1986). His most recent works produced the language Oberon, a descendant of Modula-2, which served to design the operating system with the same name (1986–1989). He was Chairman of the Division of Computer Science (Informatik) of ETH from 1982 until 1984, and again from 1988 until 1990. Since 1990, he has been head of the Institute of Computer Systems of ETH.

In 1978, Professor Wirth received Honorary Doctorates from York University, England, and the Federal Institute of Technology at Lausanne, Switzerland, in recognition of his work in the fields of programming languages and methodology. In 1983, he was awarded the Emanuel Priore prize by the IEEE, in 1984 the A. M. Turing prize by the ACM, and in 1987 the award for Outstanding Contributions to Computer Science Education by ACM. In 1987, he was awarded an Honorary Doctorate by the Universite Laval, Canada, and in 1993 by the Johannes Kepler Universitat in Linz, Austria. In 1988, he was named a Computer Pioneer by the IEEE Computer Society. In 1989, Professor Wirth was awarded the Max Petitpierre Prize for outstanding contributions made by Swiss noted abroad, and he received the Science and Technology Prize from IBM Europe. He was awarded the Marcel Benoist Prize in 1990. In 1992, he was honored as a Distinguished Alumnus of the University of California at Berkeley. He is a member of the Swiss Academy of Technical Sciences and a Foreign Associate of the U.S. Academy of Engineering.