Eidgenössische
Technische
Hochschule
Zürich

*Institut
für
Informatik*

Niklaus Wirth

Programming languages: what to demand and how to assess them

Professor Cleverbyte's visit to heaven

PROGRAMMING LANGUAGES: WHAT TO DEMAND AND HOW TO ASSESS THEM

N. Wirth, ETH Zürich

Abstract

The software inflation has led to a software crisis which has stimulated a search for better methods and tools. This includes the design of adequate system development languages.

This paper contains some hints on how such languages should be designed and proposes some criteria for judging them. It also contains suggestions for evaluating their implementations, and emphasizes that a clear distinction must be made between a language and its implementation. The paper ends with concrete figures about a Pascal implementation that may be used as yardstick for objective evaluations.

Paper presented at the Symposium on Software Engineering, Belfast, 8-9 April 1976.

PROGRAMMING LANGUAGES: WHAT TO DEMAND AND HOW TO ASSESS THEM

The cost of computing power offered by modern hardware is about 1000 times cheaper than it was 25 years ago. As a consequence, computer methods are applied to much more complicated and sophisticated problems. The result is the manufacture of very complex and large programs. In this phenomenon of Software Inflation, operating systems took the lead, but there are indications that many application oriented programs, including data management systems, are bound to become at least as large and complicated.

In their struggle to build such complex systems, in their continual fight against logical mistakes and unforeseen difficulties, against unexpected growth of code and unreached performance figures, against cost overrun and deadlines, engineers are groping for more adequate methods and tools. They range from management and programming principles to testing techniques and programming larguages. The important role of programming languages in the design of large systems is now being recognised [4]. In fact, they are indispensible. As a consequence, interest in better programming languages is revived, and industrial, governemental, and military circles are establishing committees to design such languages. The programmer and engineer is confronted with the urgent question: what should we ask of these languages, and what can we expect from them?

This paper will deal primarily with programming languages. But I am tempted to convey some observations from the hardware front that reveal a strong analogy to happenings in the area of language development.

After the first generation of computers had evolved into some truly large-scale machines, a second generation emerged, the so-called minicomputers. By that time, the larger machines were already progrmmed primarily in "higher-level" languages, such as Fortran. But the minicomputers threw programmers back into the dark age of assembly coding and bit pushing, consequently offsetting much of the cost savings in hardware by increasing cost in program preparation and upkeep. The reason for this regress was not so much the fact that the minicomputers' stores were too small to hold a compiler, but that their structure, order code, and architecture were determined in such an excruciatingly intelligent way that an automatic compiler was at a decided disadvantage compared to the cunning machine code programmer. Now we witness the emergence of the third generation of computers, the so-called micro-processors. The same phenomenon is repeating itself. Minicomputers have advanced to the state where most people realise that hand-coding is an arduous, hazardous, and costly business, and therefore prefer to use even mediocre compilers on their minis. So the old art of trickery is transferred to microprocessors, advertised, taught, and sold under a new heading: microprogramming. Again, the primary reason for this movement is the unnecessary and undesirable complexity that microprocessor designers mould into their chips during their flights of fancy. The first commercially available microprocessor is indeed of appalling baroqueness. Naturally, competitors try to outdo this very complexity, with the result that successors will in all probability be worse. Perhaps pocket calculators will repeat this story a third time.

Why don't manufacturers produce powerful but simple processors? Because complexity has proven to be a sure winner in attracting customers that are easily impressed by sophisticated gadgets. They haven't sufficiently reaslised that the additional performance of a complex design is usually much more than offset by its intransparency or even unreliability, difficulty of documentation, likelihood of misappliction, and cost in maintenance. But we shall probably have to wait for a long time, until simplicity will work as a sales argument. To be sure, "simple" must not be equated with "simple-minded", or "unsophisticated", but rather with "systematic" and "uncompromising". A simple design requires much more thought, experience, and sound judgement, the lack of which is so easily desiguised in complexity. And here we hit the source of our dilemma: a simple design that requires more development labor than a complex design isn't very attractive to a trade-secret oriented organization in a profit-oriented society.

LANGUAGES TO INSTRUCT OR TO CONSTRUCT MACHINES?

The same phenomenon is chiefly responsible for a similar development in programming languages. Here, the temptation to accumulate facilities for specialised tasks is overwhelming, and the difficulties in finding generally useful, practical, yet systematic and mathematically appealing concepts are even greater. They require wide experience, ranging from familiarity with diversified application areas, through intimate knowledge of programming techniques, to insight in the area of hardware design. Simplicity appears as even less glamorous, and the possibilities to mend and cover up defects or inconsistencies are unparallelled. The cost is enormous, when these cover-up activities have reached their limits. These costs, however, are usually carried by the customer rather than the designer.

In addition to the general gross underestimation of the difficulties of good language desgin, there appears to be a lack of understanding of its purpose. Dijkstra once remarked that most programmers adhere to the old-fashioned view that _the purpose of our programs is to instruct our machines_, whereas the modern programmer knows that _the purpoce of our machines is to execute the programs_ which represent our abstract machines. I consider both views as legitimate, depending on the circumstance. A considerable step in the right direction will be taken, when designers and programmers become actively conscious of these two views and their fundamental difference. Unfortunately, so far very few have been aware of them. Let me therefore dwell somewhat longer on this point.

It has by now become widely accepted that the primary goal of programming languages is to allow the programmer to formulate his thoughts in terms of abstractions suitable to his problem rather than in terms of facilities offered by his hardware. Yet we encounter the phenomenon that most programmers, although using higher-level languages, know the representation of their program and data in terms of computer code to a surprising level of detail. The result is that their programs often make active use of this hardware-oriented knowledge and cannot be understood without it. One is tempted to conclude that these programmers have not recognized the true objective of their language: To allow the precise, formal specification of abstract machines.

But the languages too must take part of the blame. Most programmers today start their career by learning a higher level language, for example Fortran. After a few attempts at program testing, the programmer finds out that knowledge of the computer's architecture, instruction code, and - above all - its data representation is a necessary ingredient of this profession. For, if something "unexpected" happens, the computer replies not in the programmer's language - i.e. in Fortran - but in terms of its own, which consists of cryptic words and octal or hexadecimal numbers. This leads the novice into the "real world" of computing, and he realises that the constructs properly described in his manual are but a small subset of what the computer can actually do. For example

1. logical values are represented like numbers, and space can be saved by packing many of them into one "word". Selection of individual bits can be achieved by appropriate arithmetic, since the language doesn't really know whether the data represent a set of logical values or a single number.

2. an array element with index Ø can be simulated by declaring a simple variable one line ahead of the array which starts with index 1. The zero index element can then for example be used as a sentinel in a linear search through the array.

3. the control variable in a DO statement after termination has a value which is equal to the DO-limit plus 1 (if the step is - as usual - unity).

4. a modulo operation on an integer variable by a power of 2 can be programmed by an .AND. operation (if the integer is positive!).

5. 10 characters are packed into one word and can be extracted by suitable arithmetic and .AND. operations. For instance, two such 10-tuples can be compared by a single subtraction (and the result is correct, if both operands start with a letter or a digit less than 5!).

In all these cases, the main culprit is the language that does not provide suitable constructs to represent in a proper astract way those features that the computer itself possesses. It is only natural that language designers therefore aim at introducing these facilities in newer languages. This leads to the introduction of a richer set of data structures, strings, sequences, etc., but unfortunately we also find features that are patently machine-oriented rather than corresponding to any well understood mathematical abstractions and objects. For example:

1. the label list (called switch), permitting an indexed jump, and the label variable permitting "assigned go to".

2. the address, reference, or pointer to variables and points in the program, and the use of ordinary arithmetic operations to manipulate them.

3. the interrupt as an event or "on-condition".

4. the bit-string as a set of logical values, denoted by octal numbers.

5. the Equivalence statement permitting the sharing of store for different sets of variables (supposedly used during disjoint intervals of the computation).

Now what could be wrong with these features? It is the fact that they neither help the programmer to think in terms of structures suitable to his problem, nor enable a compiler to double-check the legality of the program statements within a well-defined framework of abstraction. Instead, they merely represent structures suitable to the machine disguised in the costume of a high-level language, and they leave the task to find appropriate applications up to the programmer. Hence, the advantage of using a language with these features over using assembly code is only marginal. Perhaps it increases a programmer's productivity, if measured in lines of code per day. But the far more important task of increasing a programmer's ability to find structures most appropriate to the original problem, to find inherently effective solutions, and to design reliable programs, is affected to a much lesser degree.

In order to illustrate this subtle but important point let me offer you the following language constructs as alternatives to those critizised.

1. Instead of a label list and an indexed go to statement, introduce a selective statement. It not only eliminates the need for explicit labels and jumps, but makes the choice of precisely one of several statements much more obvious.

```
  switch S = L1, L2, L3, L4;

     goto S[i+1];                         case i of
  L1: statement-0; goto L5;                 0: statement-0;
  L2: statement-1; goto L5;                 1: statement-1;
  L3: statement-2; goto L5;                 2: statement-2;
  L4: statement-3; goto L5;                 3: statement-3
  L5: ...                                 end
```

In the above pieces of programs, one of four statements is to be executed, namely statement-0 in the case of variable i having the value 0, statment-1 in case i=1, etc. This is concisely and naturally expressed by a case statement [8]. Instead, the Algol-60 program at the left uses a goto statement referring to a switch declaration, in analogy to an indexed branch instruction in assembler code.

2. If pointers are to serve to construct lists and trees, a facility for defining recursive data structures might well replace them and express the intended data structure more naturally. For example (see Fig. 1):

```
  type list = (node: integer; tail: list)
  type tree = (node: integer; left,right: tree)
```

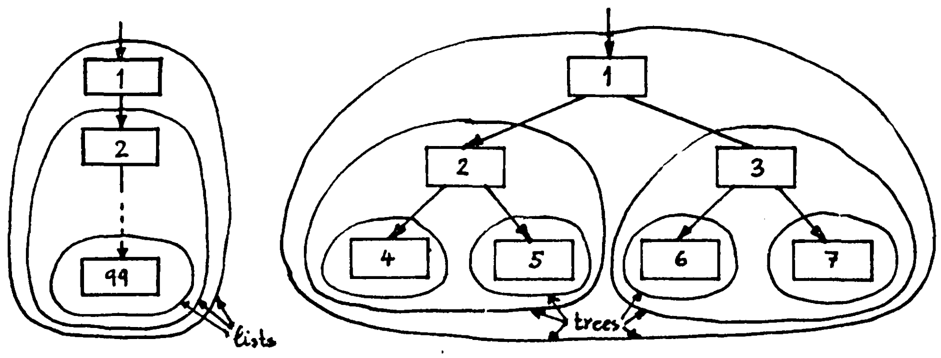

Fig. 1. Lists and trees as recursive structures

If more general structures, including rings are to be made available, or if the main objective is data sharing, then pointers should at least be restricted to the role they must play, namely to refer to other objects. All notions that suggest that a pointer is represented by an integer denoting a storage address must be avoided. If a language supports the notion of data types, each pointer should be restricted to point to one type of object only. This permits an inexpensive compile-time check to prevent many common and costly errors in the use of pointers [9]. For example:

```
DECLARE
  1 TREE_NODE CONTROLLED (CURRENT)  type treenode =
   2 KEY FIXED BINARY,                record key: integer;
   2 (LEFT,RIGHT) POINTER,              left,right: treenode
  1 LIST_NODE CONTROLLED (CURRENT)    end;
   2 KEY1 FIXED BINARY,              listnode =
   2 (NEXT,TREE) POINTER,             record key: integer;
  ROOT POINTER STATIC                   next: listnode;
                                        tree: treenode
                                      end;
                                    var root: listnode
```

The above pieces of program, expressed in PL/I at the left and Pascal at the right, allow to generate a data structure consisting of a ring of listnodes which are the roots of binary trees (see Fig. 2). The danger of the PL/I formulation lies in the circumstance that treenodes may be inserted inadvertantly in place of listnodes and vice-versa, and that a reference to one kind of node is possible under the misbelief that it is a node of the other kind. Such an error cannot even be detected at the time of node generation. In the Pascal version, this kind of confusion would already be detected at compile-time, because of the distinction of pointers to listnodes from pointers to treenodes.

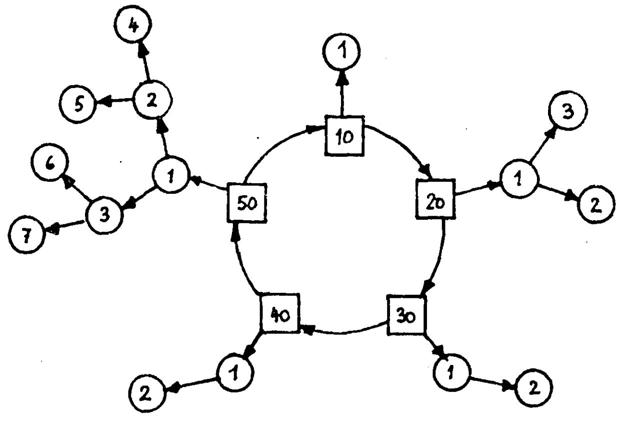

Fig.2. Ring of tree structures

3. An interrupt is a highly machine-oriented concept that allows a single processor to participate in the execution of several concurrent processes. A language should either be devoted to the formulation of strictly sequential algorithms, in which case the

interrupt has no place as a concept, or it is designed to express the concurrency of several sequential processes. In this case a suitable form of synchronization operations must be found, but again the interrupt as a concept is inappropriate, because it refers to a processor (machine) instead of a process (conceptual unit of the abstract algorithm).

4. The bitstring or word, if used as a set of logical values could well be represented as a Boolean array with indices ranging from 1 to w (where w is the wordlength of the computer). However, the denotation of constants of this type is usually by octal or hexadecimal numbers, which are conceptually foreign to the notion of logical values. A more natural concept that can very well be implemented by bitstrings is the set (of integers between 1 and w).

```
bits b                  var s: set
b := 132B;              s := {2,4,5,7}
if b[4] then            if 4 in s then
b1 and b2               s1 * s2 (set intersection)
```

5. The dangers of a facility like the Equivalence statement to share store lie not so much in the conceptual realm as in the pitfalls of its application. It is too easy to forget the fact that the different stets of variables should be mutually exclusive in time. Hence, a facility that does not necessarily advertise shared use of store, but instead implicitly allocates and frees store as needed, would be preferable by far. This is precisely the effect achieved by the facility of variable declarations local to procedures of Algol 60. It enables a compiler to guarantee that inadvertant use of the variable under the wrong name is impossible.

```
COMMON A,B
EQUIVALENCE A,B                 procedure P1(   );
SUBROUTINE S1(  )                 var a: T1;
   ... A ...                    begin ...a... end;
END
SUBROUTINE S2(  )               procedure P2(   );
   ... B ...                      var b: T2;
END                             begin ...b... end
```

I believe that there will be no real progress until programmers learn to distinguish clearly between a language (definition) and its implementation in terms of compiler and computer. The former must be understood without knowledge of the latter. And we can only expect programmers to understand this vital distinction, if language designers take the lead, and when implementors and manual writers follow that lead.

Criteria for judging a language and its documentation

Hence, we conclude that the first criterion that any future programming language must satisfy, and that prospective customers must ask for, is a complete definition without reference to compiler or computer. Such a definition will inherently be of a rather mathematical nature.

To many hardcore programmers, this demand perhaps sounds academic and (nearly) impossible. I certainly have not claimed that it is easy! I only claim that it is a necessary condition for genuine progress. I even have considerable sympathy for objections and reservations. Given a particular problem and confronted with one's installed hardware, one is often close to the point of despair when trying to maintain these high aspirations. It is therefore precisely the criterion where most language designers - often unconsciously - compromise and fail.

One may argue legitimately that there will always remain certain aspects of hardware that will be particular if not peculiar and that must be utilized and programmed as well (evidently enforcing the "old view" upon the programmer). We mainly think of interfaces to peripheral equipment, input/output devices, on-line sensors, or machinery to be controlled. But even in this area we must aim at a much higher standard of functional definition. Until this is widely achieved, language designers are well-advised to provide a facility to delineate modules within which certain device dependent language features are admitted and protected from access from elsewhere in a program. Such a facility, if well designed, would obviate the hitherto common practice of using several languages of different "levels" in designing a large system. This is a point of considerable practical importance, because interfacing between different languages (and operating systems) is precisely the occasion that most frequently forces programmers to step down to the "bit pattern level" as the only common ground of all implementations.

Hence, I recommend that a future language must provide a modularization facility which introduces and encapsulates an abstract concept. Such concepts can then be built out of concepts defined in lower level modules, and will express composite objects and actions on higher levels. This modularization facility is instrumental in keeping the size of a language - measured in terms of the number of data types, operators, control structures, etc. - within reasonable bounds. Without it, the temptation to include an additional feature each time a new application comes to mind is enormous. It can hardly be resisted, if there is no provision for expressing it in a closed and protected form within the language.

This leads us to another criterion for judging future language proposels: their size. We have witnessed the traumatic effects of languages of extreme size, whose descriptions comprise hundreds of pages of specialised and diffuse terminology with the disguised purpose to disguise confusion. A journey through the world of programming language manuals is indeed a sobering experience. The failure to distinguish between language definition and compiler description, between the meaning of language constructs and restrictions imposed by an implementation, between essential and incidential, has already been mentioned. But I must point out a common deficiency of even more fundamental nature: poor mastery of (natural) language. This phenomenon is unfortunately very widespread not only in manuals but also in the prolific computer science literature. It is not my intention to recommend the practice of embellishing imprecise thoughts with artful language, but I advise any author to straighten out his thoughts until simple words suffice to express them. In programming, we are dealing with complicated issues, and the more complicated an issue, the simpler must be the language to describe it. Sloppy use of language - be it English, German, Fortran, or PL/1 - is an unmistakable symptom of inadequacy.

Hence, our next demand addressed to future language designers is _conciseness and clarity of description_, and _sound use of language_. To give a concrete figure, the definition of a language, comprising its syntax specifying the set of well-formed sentences, and its semantics defining the meaning of these sentences, should not extend over more than 50 pages. This primary document should be accompanied by separate documents describing implementations, their limitations, effectiveness, and their reactions to ill-formed programs. The total length of these documents should be not more than 25 pages, and they _must be written in good style, devoid of ill-defined technical jargon_. Anything else is unacceptable, regardless of the high-level committees sponsoring the product, the pressing ecomonic reasons, the urging of politicians to promote international cooperation, governmental blessing, or even commercial advertisement campaigns. On the contrary, the appearance of such decor must be taken as a call for extra vigilance.

Technical criteria for judging a language implementation

My insistence on separating the language, its syntax, and its semantics as an abstract entity on the one side, and its implementation as a concrete tool on the other side, should not be interpreted as emphasis of the abstract at the expense of technical realities. We cannot close our eyes to the fact that programs are developed exclusively either to be executed by computers or as academic exercises. Hence to most people a language is at most as good as its compiler. My point is that we should not waste our time evaluating a compiler until we have closely examined the language. However, if a language has shown to be conceptually sound, what are the criteria to judge a compiler? Let me list the most important ones.

The _compiler must be totally reliable_. This requirement is three fold. First, it implies that every program is checked against _every single rule of the language_, that no formally incorrect program be allowed to pass without an indication. Second, it implies that any correct program is translated correctly. All efforts of systematic design, program verification, etc. rely on total compiler correctness. Third, no incorrect program can be allowed to cause the compiler to crash. These are very stringent conditions not only for the compiler engineer, but also for the language designer. For, under this rule the hardships of the former grow exponentially with the generosity of the latter. Consider, for example, the case where a language definition contains the rule that there may be no procedures exerting so-called side-effects on non-local variables. Then we ask that a compiler be able to detect such side-effects.

Inspite of its exhaustive checking facilities, a compiler must _compile at reasonable speed_. This is particularly important when constructing large programs, such as operating systems and compilers themselves. The figure of one second per page of source program is a reasonable figure for a medium size computer. An efficient compiler makes all desire for so-called interactive or incremental compilation disappear, and reduces the need for separate compilability of program parts significantly. If part compilation is provided, then the compiler must be able to maintain full checks for all allowed interfaces, be they parameters (type compatibility) or global variables. Otherwise part compilation is a mixed blessing.

The next requirement of a good compiler is that it generate _efficient code_. This does not mean that every single odd facility of the hardware has to be utilised at all cost. But it implies the selection of reasonably obvious code, and in particular the lack of virtually any kind of run-time support routines. A most crucial point is an effective code for procedure calls.

A related requirement is that the _execution cost_ of the code be _reasonably predictable_. There must be no circumstances where a language construct suddenly becomes drastically more expensive, if used in a certain special context. The programmer should have the possibility to understand the approximate costs of all language constructs. The same holds for the storage space consumed by code and - even more important - for data. For example, an implementation where the efficiency of access to indexed variables depends on the lower index boend being 0 or not, is rather undesirable. So is a system where the storage requirements of the two rectangular arrays

```
    a1: array[1:2, 1:1000] of integer
    a2: array[1:1000, 1:2] of integer
```

are very different.

The _compiler_ itself should also be _reasonably compact_. Bulky compilers are usually inefficient too, particularly because loading is costly and inconvenient, and because the job priority will be lower - assuming a fair scheduling policy - if a large store is requested. This point is even more essential in interactive environments, where a system's swapping activity is greatly increased by colossal compilers.

Once again, let me emphasise the feedback on language design: these requirements postulate nothing less than that the designer must be intimately familiar with all techniques and details of implementation.

A compiler must provide a _simple and effective interface to the environment_, its file system, and/or its input and output devices. This places the requirement on the language design that it should reflect such objects in a decent way. The compiler and its code should not impose any additional overhead through such an interface, as for example extra buffering of transmitted data.

All preceding requirements concern the programmer directly. There are additional ones stemming from considerations of compiler maintenance problems. One is that the _compiler be written in its own language_ (always assuming that we are concerned with a general purpose programming language). A compiler written completely in a high-level language is immeasurably easier and safer to adapt to changing environments and demands. Only such a description enables you to pinpoint possible mistakes in a short time and to correct them immediately. Moreover, it is the best guarantee that the implementor has taken care to produce a good compiler; not only because sloppy work becomes much more subject to scrutiny by customers, but also because an effort to generate efficient and compact code immediately pays off in increased performance of the compiler itself.

If a language and its compiler are both of sufficient quality to define and process themselves, it also becomes economical to abandon the concept of "binary program libraries" and to collect and retain programs in their source form alone.

All these requirements more or less directly influence the design of a language itself. They all suggest a great deal of restraint of the designer against his flights of fancy. The most important argument for this point comes from the compiler engineer; _the development cost of a compiler_ should stand in a proper relationship to the advantages gained by the use of the language. _This holds also for individual language features_. Hence, the language designer must be aware of the additional

amount of effort needed to implement a feature under the presence of various other features. Very often such costs cannot be indicated without consideration of context.
For example:

1. The cost of implementation of dynamic arrays is negligible, if arrays cannot occur as components of other structures. If they can, the problem is very much more complex.

2. Packed data structures are relatively straight-forward to implement, if all structures are static, i.e. if all their characteristics are constant and known at compile-time. The difficulties multiply, if dynamic structures are allowed, or if merely a static packed structure can be passed as a parameter to a subroutine, in which its size is unknown.

3. Implementation of sequential files becomes drastically more complex, if the file elements are allowed to vary in type (size), whereas this freedom has little effect on the complexity of compiling arrays.

Hence, a proper design is characterised equally by what is omitted as by what is included.

Can these criteria be met?

I have suggested a number of criteria by which to judge present and future language designs and implementations of them. I admit that they are rather stringent. It is important to examine them critically and, if one has agreed with them, to uphold them, even if perhaps one has to abandon some of one's pet ideas on features that a language should contain.

Postulating stiff criteria is, however, an easy matter, and practicing programmers have learned to be suspicious of academics who preach high- spirited ideals. So perhaps I owe a proof that it is indeed possible to achieve these postulated merits by a single language. I am prepared to do so by providing a few figures and facts about the programming language Pascal . I offer this language only as a yardstick, in full awareness that Pascal is not the ultimate wisdom in language design, definition, and documentation. After all, a yardstick that cannot be surpassed would ill serve as an encouragement for future efforts.

First, a brief sketch of the language: Pascal offers a set of program structuring facilities supporting the concepts of structured programming. It includes well-known forms of conditional, selective, and repetitive statements. Its subroutines can all be activated recursively, and there are several kinds of parameters: expressions (by value), variables (by reference), procedures, and functions. Variables are declared to be of a fixed type. There are the standard types integer, real, Boolean, and character. In addition, new types can be defined within the language. A scalar type is defined by enumerating its possible values, a structured type is defined by indicating its structuring scheme and the type(s) of its components. There are four basic structuring schemes: arrays, records, sets, and (sequential) files. In addition, dynamic structures of any pattern can be constructed with the aid of pointers, with comprehensive and inexpensive checks of the validity of their use. The language is defined by a concise report of 28 pages [11,14], and an attempt has been made to define its semantics by rigorous axioms [10].

Second, a brief sketch of the compiler (developed at ETH for the CDC 6000 computer family): It performs a complete check of syntax and type compatibility rules. Errors are accurately pinpointed and care is taken to avoid spurious messages. Great care is taken to generate effective code. For example

1. Registers are used in a highly efficient way.

2. Address computation of components of structured variables is performed at compile time wherever possible.

3. Multiplications and divisions by powers of 2 are implemented as shifts.

Language rules that cannot be checked at compile-time are verified at run-time. This includes checking of index bounds, of case expressions, of assignment compatibility to subrange variables, etc. Upon detection of an illegal operation, a symbolic post-mortem dump is provided, listing currently accessible variables by name and current value.

The compiler supports the data packing facility of Pascal. On a computer with large wordlength, this can well lead to savings of storage by sizeable factors (up to 60 on the CDC system). The compiler itself profits by this, although the routines to implement packed data representations are extensive and complicated.

Moreover, the compiler provides a smooth interface to the resident file system. Files used in a program and existing before and/or after execution are clearly listed as parameters in a program heading. The compiler generates standard relocatable code and allows linkage with separately compiled procedures.

The single-pass compiler requires 20000 words (= 160000 bytes) for code and data to compile small programs, and 23000 words to recompile itself. (By comparison, the standard Fortran compiler requires 20000 words.) The efficiency of the compiled code is indicated by a few sample programs in the Appendix. The average compilation speed is 110 lines of source code per second (measured when compiling the compiler). Compiling, loading, and executing the null-program takes 0.3 seconds. These figures have been obtained on a CDC 6400 computer (roughly equivalent to IBM 370/155 or Univac 1106).

The entire compiler is programmed exclusively in Pascal itself [16]. There is no assembly code interspersed in the Pascal text. Every program is supported by a small run-time routine package that provides the interface to the computer's peripheral processors and the operating system. This nucleus is programmed in assembly code and occupies just 500 words. Conversion routines for numeric input and output (including floating-point conversion) are also described fully in Pascal.

The compiler itself is about 7000 lines long. Hence, it takes only 63 seconds of processor time (on a CDC 6400) to recompile the compiler. By comparison, a cross-reference generator, also programmed entirely in Pascal, takes 30 seconds to produce an alphabetically ordered cross-reference table of the compiler program.

The latest compiler (again for the CDC 6400) was developed by a single expert programmer in 16 (full-time equivalent) months [1,2]. This figure excludes work on the small support package and the I/O conversion routines. It was developed according to rigid discipline and the top-down, stepwise refinement technique [15]. Its remarkably high reliability is primarily due to its systematic design and the use of a suitable language for coding it.

Last but not least, the language Pascal was designed <u>seven years</u> ago. The first compiler was operational in late 1970. Since then the language has undergone extensive use and scrutiny [6,12]. Sufficient practical experience is available to make an objective assessment of its utility [17], many other compilers have been or are being developed on other computers [5,13], and Pascal has already spurred further developments in the direction of multiprogramming [3].

So much about Pascal. It should suffice to convince that the afore postulated criteria are more than wishful thinking, but objectives that <u>can</u> be achieved, because they already <u>have</u> been achieved to a fair degree. My primary conclusion is that Pascal is a language which already approaches the system complexity, beyond which lies the land of diminishing returns. One should therefore be rather critical about new language proposals that usually start from scratch and rapidly build up to even greater complexity. I have provided this information and these figures in order that future languages - no matter where they come from - may be objectively compared, by the customers who will have to pay for them.

References

1. Ammann, U., The method of structured programming applied to the development of a compiler. International Computing Symposium 1973, A. Günther et al., Eds., North-Holland (1974).
2. --- Die Entwicklung eines Pascal-Compilers nach der Methode des strukturierten Programmierens. ETH-Diss. 5456 (1975).
3. Brinch Hansen, P., The programming language Concurrent Pascal, IEEE Trans. on Software Engineering 1, 2, 199-207 (1975).
4. Brooks, F.P. Jr., The Mythical Man-month. Essays on Software Engineering, Addison-Wesley, Reading (1975).
5. Friesland, G. et al., A Pascal Compiler bootstrapped on a DEC-System 10, Lecture Notes in Computer Science, 7, 101-113 (Springer-Verlag 1974).
6. Habermann, A.N., Critical comments on the programming language Pascal, Acta Informatica 3, 47-57 (1973).
7. Hoare, C.A.R., Quicksort, Computer Journal 5, 1, 10-15 (1962).
8. --- Case Expressions, Algol Bulletin 18.3.7. pp. 20-22 (Oct. 1964).
9. --- Record Handling, in "Programming Languages", F. Genuys, ed., Academic Press (1964).
10. --- and Wirth, N., An axiomatic definition of the programming language Pascal, Acta Informatica 2, 335-355 (1973).
11. Jensen, K. and Wirth, N., PASCAL - User Manual and Report, Lecture Notes in Computer Science, Vol. 18 (1974), and Springer Study Edition (1975), both Springer-Verlag.
12. Lecarme, O. and Desjardins, P., More comments on the programming language Pascal, Acta Informatica 4, 231-243 (1975).
13. Welsh, J. and Quinn, C., A Pascal compiler for the ICL 1900 series computers, Software - Practice and Experience 2, 73-77 (1972).
14. Wirth, N., The programming language Pascal, Acta Informatica 1, 35-63 (1971).
15. --- Program development by stepwise refinement, Comm. ACM 14, 4, 221-227 (April 1971).
16. --- The design of a Pascal compiler, Software - Practice and Experience 1, 309-333 (1971).
17. --- An assessment of the programming language Pascal, IEEE Trans. on Software Engineering 1, 2, 192-198 (1975), and SIGPLAN Notices 10, 6, 23-30 (1975).

Acknowledgement

The author is grateful to C.A.R.Hoare for his many helpful comments and suggestions.

APPENDIX

PASCAL Test Programs

The purpose of the following sample programs is to convey an impression of the character of the programming language Pascal, and to provide some performance figures for comparative studies. These figures were obtained on a CDC 6400 computer with the SCOPE 3.4 operating system. The statements "writeln(clock)" indicate the points where the time was taken.

1. Generate a table of powers of 2

This program computes the exact values of 2**k and 2**(-k) for k=1...n, and prints them in the form

```
 2     1   .5
 4     2   .25
 8     3   .125
16     4   .0625
32     5   .03125
64     6   .015625
..................
```

```
program powersoftwo(output);
const m = 30; n = 90;   (* m >= n*log(2) *)
var exp,i,j,l: integer;
    c,r,t: integer;
    d: array [0..m] of integer;  (*positive powers*)
    f: array [1..n] of integer;  (*negative powers*)
begin l := 0; r := 1; d[0] := 1;
   writeln(clock);
   for exp := 1 to n do
   begin (*compute and print 2**exp *)  c := 0;
     for i := 0 to l do
     begin t := 2*d[i] + c;
       if t >= 10 then
         begin d[i] := t-10; c := 1
         end
       else
         begin d[i] := t; c := 0
         end
     end ;
     if c > 0 then
       begin l := l+1; d[l] := 1
       end ;
     for i := m downto l do write(' ');
     for i := l downto 0 do write(d[i]:1);
     write(exp:5, '  .');
     (*compute and print 2**(-exp) *)
     for j := 1 to exp-1 do
     begin r := 10*r + f[j];
       f[j] := r div 2; r := r - 2*f[j]; write(f[j]:1)
    end ;
    f[exp] := 5; writeln('5'); r := 0
  end ;
  writeln(clock)
end .
```

This program uses integer arithmetic exclusively. Execution time for computing the powers of 2 (n=90) was measured as 916 (813) msec. The figure in parentheses is obtained when run-time index bound checks are disabled.

2. Palindromic squares

A number is a palindrome, if it reads the same from both ends. Find all integers between 1 and 10000 whose squares are palindromes! For example: sqr(11) = 121, sqr(22) = 484, sqr(2002) = 4008004.

```
program palindromes(output);
  var i,j,l,n,r,s: integer;
      p: boolean;
      d: array [1..10] of integer;
begin n := 0; writeln(clock);
  repeat n := n+1; s := n*n; l := 0;
    repeat l := l+1; r := s div 10;
      d[l] := s - 10*r; s := r
    until s = 0;
    i := 1; j := l;
    repeat p := d[i]=d[j];
      i := i+1; j := j-1
    until (i>=j) or not p;
    if p then writeln(n,n*n)
  until n = 10000;
  writeln(clock)
end .
```

Execution time was measured as 3466 (2695) msec.

3. Quicksort

This program sorts an array of 10000 integers according to the method called Quicksort [7]. It uses a recursive procedure. The maximum depth of recursion is ln(10000).

```
program quicksort(output);
  const n = 10000;
  var i,z: integer;
      a: array [1..n] of integer;

  procedure sort(l,r: integer);
    var i,j,x,w: integer;
  begin (*quicksort with recursion on both partitions*)
    i := l; j := r; x := a[(i+j) div 2];
    repeat
      while a[i] < x do i := i+1;
      while x < a[j] do j := j-1;
      if i <= j then
        begin w := a[i]; a[i] := a[j]; a[j] := w;
          i := i+1; j := j-1
        end
    until i > j;
    if l < j then sort(l,j);
    if i < r then sort(i,r)
  end (*sort*) ;

begin z := 1729;  (*generate random sequence*)
  for i := 1 to n do
    begin z := (131071*z) mod 2147483647; a[i] := z
    end ;
  writeln(clock);
  sort(1,n);
  writeln(clock)
end .
```

Execution time: 4098 (2861) msec.

4. Count characters in a file

The following program copies a text (file of characters) and counts the transmitted blanks, letters, digits, special symbols, and lines. It also inserts a printer control character at the beginning of each line.

```
program countcharacters(input,output);
  var ch: char;
    c0,c1,c2,c3,c4: integer;  (*counters*)
begin writeln(clock); linelimit(output, -1);
  c0 := 0; c1 := 0; c2 := 0; c3 := 0; c4 := 0;
  while not eof(input) do
  begin write(' '); c0 := c0+1;
    while not eoln(input) do
    begin read(ch); write(ch);
      if ch = ' ' then c1 := c1+1 else
      if ch in ['a'..'z'] then c2 := c2+1 else
      if ch in ['0'..'9'] then c3 := c3+1 else c4 := c4+1
    end ;
    readln; writeln
  end ;
  writeln(clock);
  writeln(c0,' lines');
  writeln(c1,' blanks');
  writeln(c2,' letters');
  writeln(c3,' digits');
  writeln(c4,' special characters');
  writeln(clock)
end .
```

Execution time was measured as 4345 msec for a file with 1794 lines, 23441 blanks, 27331 letters, 17Ø5 digits, and 9516 special characters. This results in an average of Ø.Ø68 msec per character, or 1468Ø characters per second.

5. Inout and outout of numbers

The next sample program generates a file f of 25ØØØ real numbers, and computes their sum s. Then the file is reset and read, and a checksum is computed.

```
program numericIO(f,output);
  const n = 25ØØØ; d = Ø.12345;
  var i: integer; x,s: real;
      f: file of real;
begin writeln(clock);
  x := 1.Ø; s := Ø; rewrite(f);
  for i := 1 to n do
    begin write(f,x); s := s+x; x := x+d
    end ;
  writeln(clock, s);
  reset(f); s := Ø;
  while not eof(f) do
    begin read(f,x); s := s+x
    end ;
  writeln(clock, s)
end .
```

It took 123Ø msec to generate the file, and 98Ø msec to read it. This corresponds to 49 usec to write, and 39 usec to read per number.
The amount of time increases drastically, if a decimal representation of the numbers on the file is requested. This is easily accomplished, namely by declaring the file to consist of characters instead of real numbers:

```
        f: file of char
```

In this case, the read and write statements include a conversion operation from decimal to binary and vice-versa. Generating the file then takes 28185 msec, reading takes 3Ø313 msec. This corresponds to an increase by a factor of 23 in writing and 31 in reading. (Each number is represented by 22 characters on the file).

6. Eight Queens

This program finds all 92 positions of 8 queens on a chessboard such that no queen checks another queen [15]. The backtracking algorithm is recursive.

```
program eightqueens(output);
var i : integer;
    a : array [ 1..8 ] of boolean;
    b : array [ 2..16] of boolean;
    c : array [-7..7 ] of boolean;
    x : array [ 1..8 ] of integer;
    safe : boolean;

   procedure print;
      var k: integer;
   begin write(' ');
      for k := 1 to 8 do write(x[k]:2);
      writeln
   end ;

procedure trycol(j : integer);
   var i : integer;

   procedure setqueen;
   begin a[i] := false; b[i+j] := false; c[i-j] := false
   end ;

   procedure removequeen;
   begin a[i] := true; b[i+j] := true; c[i-j] := true
   end ;

      repeat i := i+1; safe := a[i] and b[i+j] and c[i-j];
         if safe then
         begin setqueen; x[j] := i;
            if j < 8 then trycol(j+1) else print;
            removequeen
         end
      until i = 8
end;

begin for i := 1 to 8 do a[i] := true;
      for i := 2 to 16 do b[i] := true;
      for i := -7 to 7 do c[i] := true;
   writeln(clock); trycol(1); writeln(clock)
end.
```

Run-time: 1Ø17 (679) msec.

7. Prime numbers

Program "primes" computes the first 1000 prime numbers, and writes them in a table with 20 numbers per line. This takes 1347 (1061) msec.

```
program primes(output);
const n = 1000; n1 = 33;    (*n1 = sqrt(n)*)
var i,k,x,inc,lim,square,l: integer;
    prim: boolean;
    p,v: array [1..n1] of integer;
begin writeln(clock);
   write(2:6, 3:6); l := 2;
   x := 1; inc := 4; lim := 1; square := 9;
   for i := 3 to n do
   begin (*find next prime*)
      repeat x := x+inc; inc := 6-inc;
         if square <= x then
            begin lim := lim+1;
               v[lim] := square; square := sqr(p[lim+1])
            end ;
         k := 2; prim := true;
         while prim and (k<lim) do
         begin k := k+1;
            if v[k] < x then v[k] := v[k] + 2*p[k] ;
            prim := x <> v[k]
         end
      until prim;
      if i <= n1 then p[i] := x;
      write(x:6); l := l+1;
      if l = 20 then
         begin writeln; l := 0
         end
   end ;
   writeln; writeln(clock)
end .
```

8. Ancestor

The last sample program operates on a Boolean matrix. In its first part it generates a matrix r. Let r[i,j] mean "individual i is a parent of individual j". At completion of the second part, r[i,j] means "individual i is an ancestor of individual j". In the third part, the matrix is output.

```
program ancestor(output);
(*R.W.Floyd: 'Ancestor', Comm.ACM 6-62 and 3-63, Alg.96*)
  const n = 100;
  var i,j,k: integer;
      r: array [1..n, 1..n] of boolean;
begin (* r[i,j] = "i is a parent of j"*)
  for i := 1 to n do
    for j := 1 to n do r[i,j] := false;
  for i := 1 to n do
    if i mod 1Ø <> Ø then r[i,i+1] := true;
  writeln(clock);
  for i := 1 to n do
    for j := 1 to n do
      if r[j,i] then
        for k := 1 to n do
          if r[i,k] then r[j,k] := true;
  writeln(clock);
  for i := 1 to n do
  begin write(' ');
    for j := 1 to n do write(chr(ord(r[i,j])+ord('Ø')));
    writeln
  end ;
  writeln(clock)
end .
```

It takes 291 msec to generate the matrix, 1667 msec to execute the ancestor algorithm, and 578 msec to output the matrix. Since the matrix consists of 1ØØ * 1ØØ elements, 1ØØØØ (6Ø-bit) words of store are needed.
If r is declared as

```
          r: packed array [1..n, 1..n] of Boolean
```

then the required store is only 2ØØ words, or 5Ø times less. The execution times are then 4Ø6 msec to generate, 2126 msec to computed, and 642 msec to output the matrix. This is only 1.3 times more than in the case of the unpacked matrix representation.
A second version of the algorithm uses the Pascal set structure instead of a Boolean matrix. The relation r[i,j] is expressed as "j in r[i]". Since the Pascal 6ØØØ-3.4 compiler restricts sets to have at most 59 elements, the following performance comparison is based on the case n = 5Ø.

```
program ancestor(output);
(*ancestor algorithm using sets instead of boolean matrix*)
  const n = 5Ø;
  var i,j: integer;
      r: array [1..n] of set of 1..n;
begin (* j in r[i] = "i is a parent of j"*)
  for i := 1 to n do
    if i mod 1Ø <> Ø then r[i] := [i+1] else r[i] := [];
  writeln(clock);
  for i := 1 to n do
    for j := 1 to n do
      if i in r[j] then r[j] := r[i]+r[j];
  writeln(clock);
  for i := 1 to n do
  begin write(' ');
    for j := 1 to n do
      if j in r[i] then write('1') else write('.');
    writeln
  end ;
  writeln(clock)
end .
```

This program requires only 58 msec to compute the ancestor matrix, compared to 341 msec for the version using a packed array. This is a gain by a factor of 5.9.

Extract from

PROFESSOR CLEVERBYTE'S VISIT TO HEAVEN

N. Wirth, ETH Zürich

Abstract

The following fable is a grotesque extrapolation of past and current trends in the design of computer hardware and software. It is intended to raise the uncomfortable question whether these trends signify real progress or not, and suggests that there may exist sensible Limits of Growth for software too.

When I had been dead for several weeks, I began to get a little anxious. I had been hovering around, first experimenting with my novel facilities and freedom from all earthly limitations. Perhaps I ought to mention at this point that I had been a manager of a software house, and my decease had been a direct consequence of our decision to introduce both a new programming language and a new operating system at the same time. The ensuing difficulties were enormous and responsible for my spending the rest of my life on the job.

So I was disappointed to see how little difference my absence made, in spite of the fact that I had been the only one intimately familiar with all the details of these new systems. I realized that a little more or less confusion didn't really matter.

Hence I became anxious to direct my course upwards. Fortunately I remembered the report of Mark Twain's Captain Stormfield , and therefore was neither surprised by my exhilarating rush through space, nor did I expect to enter a heaven of eternal bliss. But

I expected that it would be a place of unlimited opportunities where nothing was impocsible. This expectation is, of course, quite typical of a man from the software profession.

Heaven is a complex place, and it is also astonishingly modern: I was taken aback to discover large boards with light-displays and computer terminals used to find your present location as well as the shortest path to any desired location or department. The boards list all possible subjects you may think of. They continually expand as new departments with imaginative names emerge, one about every second. I readily found Software Engineering - merely the o had been misspelled as an a, perhaps by a German clerk - and I headed off in its direction. As a new department, it was located at heaven's periphery, and I marched for several days.

When I finally reached my blessed destination with sore feet, I found the quarters almost deserted. But as luck would have it, shortly thereafter I spotted a man carrying a deck of punched cards. I was overjoyed when I recognised him as my old friend Jonathan Flagbit who several years ago had switched from computing to life insurance. "You here, inspite of all!" I exclaimed; "You don't seem to have kept up with progress" I sneered referring to his card deck.

"Don't jump to rash conclusions, Cleverbyte, I've gone through all the stages up here, and we've got he most modern equipment you haven't even dreamt of".

Being quite excited at this prospect, I asked: "May I see your modern equipment?"

"Of course you may, everything is possible up in heaven and even more so in the Software Department ofer there. All you need is to make a wish, and it shall be fulfilled".

I told him grudgingly that I could have spared my sore feet had I known this beforehand, and he replied:

"Every newcomer indulges in wishing, but soon they get tired of it. It's deceiving in the long run. Too often there are small bugs, and you get something different. So wishing isn't as wonderful as it first sounds."

I was pondering about this point, then decided that I wasn't really eager to admire their equipment. Instead, I asked: "What about programming languages?"

"That is a huge department of its own. We use thousands of languages, and some of them are so sophisticated that no amount of paper woeld suffice to hold a complete listing, so they are permanently kept on Womm, and you enquire only about what you need at the moment".

"What is Womm?" I asked, now suddenly being aware that it was me who was behind. But Flagbit didn't scorn my ignorance, or at least he concealed it magnificently and replied:

"That is our new word organising mass memory device. It is the first of its kind having an infinite capacity. Its access speed and transfer rates are still slow, but they are working on it. It has revolutionised our entire business and opened the door to a new generation of programming languages".

"I bet. But, I beg your pardon for asking, what are the goals in designing all these languages? After all, languages were invented to raise the quality, reliability, efficiency of systems, and to reduce the cost of their production", I suggested cautiously.

"Now, come on, Cleverbyte! That sounds pretty old-fashioned, even by earthly standards! I reckon you had a problem with unemployment lately too; up here it is one of major proportions. To be quite frank, it is directly responsible for the software explosion. Producing languages to make programming easier and simpler would be counterproductive. On the contrary, these languages are ideally suited to keep uncounted people on their intellectual toes, content and busy, and to maintain an image of progress and sophistication. We have whole armies of clerks writing manuals; and they love it".

I wasn't quite prepared for a sermon of such length, and it took me some time to digest this philosophy. So I asked naively:

"But have you discovered a way to comprehend these languages and profit by their use?"

"One never understands the whole thing. It is another of those stifling high-brow dogmas that one should be able to understand The Whole. When yoe are to solve a problem, you study the relevant sections of your language, and if you can't follow it, you take a course or have somebody write another manual for you. There are lots of souls waiting for attractive suggestions to teach a course or write a manual. Naturally, this will take too long if you have a genuine desire to get some problem really solved. Then you go back to first principles and simple means - just look at my cards! But it takes people a long time to find this out, just as with loosing their illusions about wishing".

"And how do you think this will be in the future? More mountains of manuals?"

"We don't really worry about the future, but if you care to know, just make a wish to be transferred to some language design committee", Flagbit remarked. We agreed that this was the best way to obtain a representative picture, for Flagbit had assured me that in the future all this was going to be done by committee. There followed a slight tremor, we were whisked away and found ourselves in the midst of a select group of obvious experts in full action. The scenery was splendid, a phantastic combination of seaside and mountain resort, making it particularly difficult for me to follow the subsequent discussion.

W: "The problem is one of coercions rather than types".
H: "Coercions can give one an amount of uniform reference which is beneficial".
I: "But the semantics change with uniform reference, that is, one has punning".
G: "Visibility is important and it must be taught as a practical concept".
I: "Visibility is conceptually hard".
D: "Visibility is tough, because of its interaction with block structure".
I: "Let us now discuss partial visibility!"
L: "A variable is like a capability".
G: "To beliebe that every variable is a reference is inaccurate. If we have sorted out visibility, then partial visibility will be easy".
R: "Algol 68 has the notion of possessing and referencing".
K: "A name cannot possess a reference!"
G: "All visibility should be coupled to compilation units".

I soon got restless for I could hardly perceive that they were talking about our subject at all. I was just about to voice my complaints when all of a sudden the whole region fairly rocked under the crash of four thousand and ninetysix thunder blasts.

"There, that's the Professor!" Flaggy whispered.

"Then let's be moving along", I urged, being anxious to leave this committee where I felt uncomfortably incompetent.

"Keep your seat, Cleverbyte", Flagbit said, "he is only just telegraphed".

"How?!"

"These blasts only mean that he has been sighted by our computerised early warning radar system. He is just off Cape Canaveral . The committee will now go down to meet him and escort him in. But he is still millions of miles away, so the show won't come up for a considerable time, yet".

We walked down to the Conference Center at leasure. I was truly amazed by its sheer size: onehundredandthirtyonethousand and seventy two seats, virtually all empty.

"This venue looks pretty deserted. I bet there is a hitch somewhere again", I remarked pessimistically. "Is he perhaps going to give one of those highly specialised talkc that only a handful of experts can follow?"

"Don't you fret, Cleverbyte, it's all right, you shall see. Of course he is going to talk specialised. That is important to maintain the proper image. But the topic is just for show anyway. People will come for social and commercial reasons; it is fashionable to have been here, and you meet friends".

At his moment there was another big bang, like that of a new supernova.

"The Professor is through the security checks now and will be on stage within seconds", Flagbit explained. And then there was a big flash, the whole place was splendidly lit up, and all the seats were suddenly taken. My chin dropped a few inches by surprise, and my friend commented with undisguised pride:

"That's the way we do it up here. Nobody worrying about being late, nobody sneaking in after the curtains went up. Wishing is quicker than walking!"

However, I spotted a slight disturbance not too far from us. Somebody was making a distinct fuss.

"What's going on down there, Flaggy?" I asked, finding this scene somewaht unusual in such a well-organised place.

"You see, since all people wish to be in the stadium when they hear the bang, computation of seat assignments presents a few problems", he remarked with calculated understatement and his pride had visibly diminished. Then he continued:

"They (his previous We now had become a deferential They!) have recently put a new supercomputer into operation, but occasionally there is still a glitch in the algorithm, although it was announced to be formally verified. It doesn't take long until you realise that every verification is worthless as long as it is itself not verified. And now this real-time algorithm is particularly sophisticated. It made great headlines under the name 'Seat assignment of the fly'".

After the excellent talk on "Quaternionic complexity on a three-tape Turing-machine without pointers" we had a drink and relaxed in the lobby. Our glasses not yet empty, a hefty, square-jawed man approached our table. Flagbit jumped up and pulled me by my sleeve.

"This man is the chief brain of our supercomputer. Hello, Megachip, let me introduce you to my friend Cleverbyte who has just arrived! What are the latest figures on your machine?"

"Well, it now works with three times the speed of light. Sixteen billion active elements placed on 2.56 million single chips!" was his reply.

By that time, I had already learned to keep my composure when hearing of staggering innovations, but nevertheless I must have been looking pretty foolish, for Flagbit interjected:

"You must know, Cleverbyte, Megachip has had the greatest idea ever: by making chips work faster than light you can read out your computed results virtually before you insert your data, provided you position your output station at a location remote from your input device".

I was at a loss for words and could merely state the obvious: "But this must revolutionize the entire computing business, particularly programming". Megachip laughed heartily:

"It sure does! All this craze about optimization is over. We have a store of several gigabytes, hundredtwentyeight thousand parallel microprocessors, sixteen thousand data channels running at megabaud rates. The whole hardware merely costs eight million pounds, which I am sure is not more than a handful of shillings was in your earthly days".

"This is a staggering feat indeed; but does your software stand up to these measures? I am sure its cost was immeasurably larger", I commented.

"The biggest single piece of software ever developed! The operating system alone takes over one billion bytes of instructions, and together with the compilers it took seventeen hundred man-centuries to develop, in spite of our loss of interest in optimization. Most of the work went into maintenance, and after several breakthroughs in reliability we now have only about 50 breakdowns per second. The real turning-point was the acceptance of the fact that a perfect, faultless system would never materialise, but that instead we had to work towards a fault-tolerant, self-recovering system. This resulted in close to 100% of the breakdowns being recovered by the system without intervention".

"These are truly awesome figures to me! But may I ask you, Mr. Megachip, how this tremendous system was developed, and in particular what programming language you use, for with the ones that I know such a feat would have been utterly impossible".

"Well, Cleverbyte, you are right, but not quite. The language used has gradually evolved by extensions in all possible directions. To give you a measure, the manuals measure threehundred thousand volumes and the compiler uses up half a billion bytes. You see, we are quite willing to let anyone keep his habits of programming; hence the language must be compatible with pretty well every previous language that has ever existed".

This decidedly began to amuse me, for I felt I had heard these arguments before. Tongue in cheek I asked:

"But doesn't this inflation of languages largely offset the gains you have made in the development of hardware? I am convinced that this diversity and the systematic retention of old mistakes is a wasteful deadweight for men and machines alike".

"Of course it is; but listen, dear Cleverbyte, we sorely need this deadweight! You just take the wrong viewpoint; this bulky software does <u>not offset</u> our hardware innovations, but <u>justifies</u> them. By Jove, how could we otherwise find any motivation in our continued hard labor? Just think of it, we are in heaven where time is eternity and speed doesn't truly matter!"