ON THE DESIGN OF PROGRAMMING LANGUAGES*

N. WIRTH

This paper reports on some past experiments in the design of programming languages. It presents the view that a language should be simple, and that simplicity must be achieved by transperence and clarity of its features and by a regular structure, rather than by utmost conciseness and unwanted generality. The paper contains an overview of the language designer's problems and dilemmas, and ends with some hints drawn from past experience.

In order to prevent misunderstanding or even disappointment, I should like to warn the reader not to interpret this title as an announcement of a general critique of commonly used languages. Although this might be a very entertaining subject, probably all that can be said about it has been said and heard by those willing to listen. There is no reason to repeat it now. Neither do I intend to present an objective assessment of the general situation in the development of programming languages, the technical trends, the commercial influences, and the psychology of their users. The activities in this field were enormous over the past years, and it would be presumptuous to believe that a single person could present a comprehensive, objective picture. Moreover, many aspects have been aptly reviewed and commented elsewhere (1-3).

Instead, I should like to convey a view of the development of design attitudes, of the shift of emphasis in design goals over the past decade. Also, I will try to provide some insight into the multitude of problems aspects, and demands that the designer of a language is facing, and to woo gently for recognition of the difficulties of this profession. Let me start by recalling some of my own reminiscences of incidents that caused me to end up in this role of language designer.

AN EXCURSION INTO "HISTORY"

My interest in computers had been awakened when I was an engineering student in the late 1950s; I was fascinated on the one hand by simplicity and inherent reliability of the basic building blocks, the digital circuits, and on the other hand by the astounding variety of effects that could be obtained by combining many of them in different ways. Moreover, these combinations could be varied not by extensive and cumbersome use of the soldering iron, but by merely composing ingenious sequences of hexadecimal digits placed on a magnetic drum. As the general task of an engineer is the improvement of his technical gadgets, I perceived that computers were an ideal ground for engineering activities, and felt that there was ample room for further improvement. This assessment turned out to be correct up to the present day. But how was progress to be achieved? One way was by enhancing the reliability and effectiveness of their electronic components. The other - hazily defined path - seemed to be to make computers more conveniently usable, to make them less of an exclusive domain of the highly trained specialist.

This was the situation when I entered graduate school at Berkeley: In one room there stood a huge prototype of a computer with a bewildering number of wires and tubes. In a much smaller room nearby there were a few such specialists, talking about a "language" and a "translator". Luckily for me, a student helping to bring the big monster of a computer into operation didn't report too enthusiastically about that project, and I perceived the feeling that the future of computer hardware design did not lie in a university department anyway. Hence, I decided to explore the other alley, although that group was surrounded by skepticism, as word passed that some of these people did neither know Ohm's law nor Maxwell's equations.

The new and fascinating project under the direction of H.D. Huskey consisted in adding facilities to the programming language NELIAC by extending the translator program. This program was, remarkably, coded in the very language that it was compiling and, in retrospect, quite advanced for its time (4). Indeed, the fascination of the project originated much more from the sense of adventurousness than from the satisfaction of achieved perfection. For, although programming in NELIAC proved to be considerably more convenient than exercises in the cryptology of hexadecimal codes, the room for still further improvements continued to appear unlimited. Looking at it from the distance, that compiler was a horror; the excitement came precisely from having insight into a machinery that nobody understood fully. There was the distinct air of sorcery.

Then Algol 60 appeared in the literature and - after the difficulties of learning the new syntactic formalism were mastered - began to provide some relief from the oppressive feeling that there was no way to bring order into large languages and compilers. Yet even then, the task of constructing an Algol

*Reprinted from *Proc. IFIP Congress 74*, 386-393, North-Holland, Amsterdam, North-Holland Publishing Company.

The author is with the Institut für Informatik, Eidg. Technische Hochschule, Zurich, Switzerland.

compiler generating acceptably good code appeared enormous. The more the Algol compiler project neared completion, the more vanished order and clarity of purpose. It was then that I clearly felt the distinct yearning for simplicity for the first time. I became convinced that we should learn to master simpler tasks before tackling big ones, and that we need to be equipped with much better linguistic and mental tools. But apparently "useful" languages had to be big.

SIMPLICITY IN GENERALITY

In this situation, A. van Wijngaarden appeared like a prophet with his idea of Generalised Algol (5). His point was that languages were not only too complex, but due to this very complexity also too restrictive. "In order that a language be powerful and elegant it should not contain many concepts and it should not be defined with many words." The new trend was to discover the fundamental concepts of algorithms, to extract them from their various incarnations in different language features, and to present them in a pure, distilled form, free from arbitrary and restrictive rules of applicability.

The following short example may illustrate the principle. In Algol 60, the concept of a subprogram appears in two features: as declared procedure, and as name parameter to procedures. The passing of a parameter is realised as an assignment of an object (the actual parameter) to a variable (the formal parameter). A simplification and concurrent increase in power and flexibility of Algol can therefore be obtained by unifying the notions of procedure and parameter, by letting them become objects that can be assigned to variables like numbers or logical values. Let such an object be denoted by the program text enclosed by quote marks; the correspondence between constructs of Algol 60 and the generalised notation are shown by the following table:

Algol 60	Generalisation
procedure P; <statement>	P := '<statement>'
Q(<expression>)	Q('<expression>')

The gain in simplicity of language is obvious and the resulting gain in flexibility is striking. For example, it is now possible to assign different subroutines to a variable at different times, or even to replace a function subroutine by a constant! We are suddenly offered the power so far reserved to the assembly language coder letting his program modify some of its own instructions.

Implementation of such far reaching generalisations were an open challenge. I decided to investigate, whether these concepts could be condensed into a minimal language and compiler, as a language without compiler seemed to be of marginal value to me. This effort resulted in my first programming language, called Euler (5). It was a success in several ways, certainly if measured by the number of subsequent implementations on a wide variety of computers. The language was accepted as an intellectual challenge, a flexible and powerful vehicle. Its simplicity and compactness made it an ideal implementation exercise for many prospective compiler engineers. Its greatest value, however, lay in revealing how simplicity should not be understood and achieved.

The premise that a language should not be burdened by (syntactical) rules that define meaningful texts (5) led to a language where it was difficult and almost impossible to detect a flaw in the logic of a program. It led to what I like to call a high-level Turing machine. Making mistakes is human (particularly in programming), and we need all the help possible to avoid committing them. But how can one expect a language to aid in avoiding mistakes, if it is even incapable of assisting in their detection.

The lesson is, then, that if we try to achieve simplicity through generality of language we may end up with programs that through their very conciseness and lack of redundancy elude our limited intellectual grasp. It is a mistake to consider the prime characteristic of high-level languages to be that they allow to express programs merely in their shortest possible form and in terms of letters, words, and mathematical symbols instead of coded numbers. Instead, the language is to provide a framework of abstractions and structures that are appropriately adapted to our mental habits, capabilities, and limitations. The "distance" of these abstractions from the actual realisation in terms of a computer is an established measure for the "height" of a language's level.

The key, then, lies not so much in minimising the number of basic features of a language, but rather in keeping the included facilities simple to understand in all their consequences of usage and free from unexpected interactions when they are combined. A form must be found for these facilities which is convenient to remember and intuitively clear to a programmer, and which acts as a natural guidance in the formulation of his ideas. The language should not be burdened with syntactical rules, it must be supported by them. They must therefore be purposeful, and prohibit the construction of ambiguities. It is a good idea to employ adequate, concise key words, and to forbid that they can be used in any other way. Prolixity is to be avoided, as it introduces a wrong kind of redundancy.

LEVELS OF ABSTRACTIONS

One of the most crucial steps in the design of a language is the choice of the abstraction upon which programs are to base. They can be selected only if the designer has a clear picture of the purpose of his language, of the area of its intended application. But all too often that purpose is not neatly specified and includes so many diverse aspects that a designer is given only inadequate guidance from prospective users. But at least he should restrict his selection to abstractions from the same level which are in some sense compatible with each other. I should like to offer three examples to this topic.

Algol 60 has chosen a well-defined set of abstract objects of computation: numbers and logical values, replacing bits and words as used on a lower level. The operations that are applicable to them are governed by mathematical laws and axioms which can be understood without referring to the number's representation in terms of bits and words. In the realm of control structures, the language introduces the operations of selective execution and repetition in the form of neatly structured statements. But their form was not sufficiently flexible - e.g. repetition is intimately coupled with a variable progressing through an arithmetic series of values, and selection is only provided among two alternatives. To provide the user with a facility for cases not covered by these control structures, the designers of Algol resorted to borrowing the universally applicable jump order from the lower level of machine coding. The goto statement is but a polished form of the jump.

This may not seem too serious in itself. But consider that now the programmer is able to use these facilities combined. The following example - which an honest Algol programmer will refrain from using, or otherwise will at least get a bad conscience - shows the point.

```
for i := 1 step 1 until 100 do
    begin S; if p then goto L end
```

The whole purpose of the for clause is to proclamate to the reader: "the qualified statement is going to be executed once for every i = 1,2,3,...,100". The jump, however, may sneakily cause this promise to be broken! Whereas jumping out of a substructure may be considered to be a matter of morale only, jumping back into a structure even raises technical problems. They require the establishment of protective rules and restrictions which are afflicted by the stigma of improvisation and afterthought. They complicate the language by burdening its definition, its comprehension, and its compiler.

The second example concerns the notion of pointers or references in high-level languages. When programming in assembly code, probably the most powerful pitfall is the possibility to compute the address of a storage cell that is to be changed. The effective address may range over the entire store (and even be that of the instruction itself). A very essential feature of high-level languages is that they permit a conceptual dissection of the store into disjoint parts by declaring distinct variables. The programmer may then rely on the assertion that every assignment affects only that variable which explicitly appears to the left of the assignment operator in his program. He may then focus his attention to the change of that single variable, whereas in machine coding he always has - in principle - to consider the entire store as the state of the computation. The necessary prerequisite for being able to think in terms of safely independent variables is of course the condition that no part of the store may assume more than a single name. Whereas this highly desirable property is sacrificed in Fortran by the use of the "equivalence" statement, Algol loses it through its generality of parameter mechanism and rule of scope. Even if the sensible rule is observed that a procedure's parameters must denote disjoint variables, one and the same variable may be referred to under more than one name, as shown by the following example.

```
begin integer a;
   procedure S(x); integer x;
   begin a := a+1; x := x*2
   end;
   a := 1; S(a); write(a)
end
```

This design partly stems from the failure to separate the roles of textual abbreviation and of parametric program decomposition, both projected onto the same facility of the procedure. For mere textual abbreviations, one might be more willing to refrain from the use of parameters; in the case of program decomposition, communication with the environment might advantageously be restricted to explicit parameters. Hence, the notion of simplicity and frugality was rather counterproductive when viewed from the point of programming security.

We note that a parameter substitution represents an assignment of the storage address of the actual variable (a) to the formal parameter (x). In the spirit of generalisation it appeared as highly logical to admit the address into the society of computable objects. The address of a variable a - now called a reference - was thus introduced in the language Euler and denoted by @a. It can be assigned to any other variable, say x. The variable a is then openly available under two names, a and x., and there is no limit to the number of further names that can be given to a. This experiment of reopening Pandora's box of storage addresses in Euler provided a clear warning of their undesirability, but didn't prevent the introduction of references into Algol 68 in their fullest flexibility.

The concept of data type provides added security insofar as a compiler may check against inadvertant use of incompatible variables in, for instance, assignments. This is, in itself, introducing another restriction unknown at the level of machine code, where every cell can be loaded with a copy of every other cell's content. The idea of simplicity through generality again led to the elimination of restrictive type rules, and of the data type in general. It was adopted by a family of languages designed to fill the apparent gap between high-level languages and assembly code, which are now known as machine oriented higher order languages (Mohols). They permit the construction of expressions with "untyped" operands and the application of both arithmetic and logical operations on the same variables. The result of programs written in such languages can only be understood through knowledge of the particular storage representation of data in terms of bits and words. Although effects may be produced that turn out to be the desired ones, they force the programmer to leave the realm of abstraction that a language is pretending to offer him.

So much for the third example of transgression of levels of abstraction in languages.

THE THREE EXAMPLES REVISITED

I do not deny that there are situations in programming which call for facilities not present in Algol-like languages. But they should not be met by compromising on the level of abstraction. The sneaky reintroduction of patently pernicious facilities from the era of machine coding is not an acceptable solution. In order to establish remedies, we must discover the true reasons for the programmers wish of such facilities. The language designer must not ask "what do you want?", but rather "how does your problem arise?" For, the answer to the first question will inevitably be "jumps, type-less operands, and addresses".

To convey an idea of the spirit in which a designer ought to approach such problems, I will sketch possible solutions to the three mentioned cases. The presented features do not provide the full flexibility inherent in jumps, type-less operands, and free address manipulation. But this is precisely their virtue; they still impose certain sensible restrictions of usage, and help maintain a programming discipline. In particular, they do not compromise the language's high level of abstraction; they do not introduce notions which can be explained only in terms of an underlying machine.

In the case of the for statement, what the programmer really needs is not necessarily a jump order, but merely a more flexible way to express termination of a repetition. The solution consists in providing a simpler form of repetitive statement whose termination does not necessarily depend on a variable moving through an arithmetic progression. Widely accepted forms are the while- and repeat statements, for example

```
while B do S
repeat S until B
```

The important point is that now the total effect of the composite statement can be deduced solely from the properties of the repeated component. The pertinent deduction rule can be formally expressed, for instance in Hoare's formalism (7). If P and Q denote any assertions on the state of the computation, then P{S}Q means: if P holds before the execution of S, and this execution terminates, then Q holds after termination. The two deduction rules governing the above statement forms are (8):

$$\frac{P \wedge B \ \{S\} \ P}{P \{\textbf{while}\ B\ \textbf{do}\ S\}\ P \wedge \neg B}$$

$$\frac{P\{S\}Q \ , \ \neg B \wedge Q \{S\} Q}{P\{\textbf{repeat}\ S\ \textbf{until}\ B\}\ Q \wedge B}$$

Another situation requiring the use of a jump in Algol arises when one statement has to be selected among many. A feature invented by Hoare in exactly the same spirit, that not only replaces a jump and a switch declaration, but expresses the selection of one case among many in a structured, orderly way, is the case statement (9).

As for the second example, one may ask why addresses or pointers are needed anyway. I do not intend to pursue this argument here, but claim that if one consents to their necessity, they should be admitted only in a considerably tamed form. Security in pointer handling can be improved drastically by the following measures:

1. Every pointer variable is allowed to point to objects of a single type only (or to none); it is said to be bound to that type. This rule allows to maintain a compiler's capability of full type checking.

2. Pointers may only refer to variables that have no explicit name declared in the program, that is, they point exclusively to anonymous variables allocated when needed during execution. This rule protects the programmer from the dangers arising when variables are accessible under different names.

3. The programmer must explicitly specify whether he refers to a pointer itself or to the object to which the pointer refers (no automatic "coercion"). This rule helps to avoid ambiguous constructs and complicated default conventions liable to misunderstanding.

The reasons why programmers sometimes wish to deal with type-less variables are more difficult to pinpoint. The most frequent one is probably the necessity to pack different kinds of data densely into a single word, which the available language always regards as an indivisible entity. For instance, we might have to pack a triple r - say a file descriptor - consisting of a name x of 6 characters, a 5-bit status information s, and a 2-digit length count n. (The 5 status bits may, for example, indicate a tape's loadpoint, end of tape, and end of record positions, its density mode and parity check status.) A pictorial representation might be

and a common way to denote the value of such a triple is as an octal (or hexadecimal) number, because the word is available in the language under the misnomer "integer". In order to determine this number, the programmer must forget his original abstractions and perform the binary encoding "by hand". If he is lucky, he obtains

$r = 0102030405064331_8$ or $r = 0420C41468C9_{16}$

Part of the true information is arithmetic in its nature (n), another is logical (s), and a third alphabetic (x). Hence, all kinds

of orders must be applicable. But this is only possible, if the operand is not restricted by its characterisation through an associated type. What the programmer really needs in this case is a data structuring facility relieving him from the tedious and errorprone labor of data encoding and packing.

As an illustration, in the programming language Pascal the triple r can be directly declared as a structured variable, yielding the dense packing indicated by the picture above (10). The first component of r is declared to be an array of 6 characters, the second to be a set of status indicators, and the third a number in the range of 0 to 99.

```
type string = packed array[0..5] of char;
  indicator = (loadpoint,eof,eor,pchk,
               highdensity);
var r: packed record
         x: string;
         s: set of indicator;
         n: 0 .. 99
       end
```

Assignments, instead of involving obscure arithmetic operations expressing shifts etc., are simply written as

```
r.x := 'ABCDEF';
r.s := [loadpoint,highdensity];
r.n := 89
```

Each of the three components has a distinct name and a distinct type. Its proper usage can be completely checked by compiler and program reader alike. Naturally, such a structuring facility complicates a compiler considerably, much depending on the quality of the underlying hardware architecture. It even increases the "volume" of a language; but significantly, it does not reduce its conceptual simplicity.

The proposed solutions to the three mentioned problem areas lie in introducing restrictive rules, and are contrary to the spirit of "power through simplicity" and "simplicity through generality". But they have already proven to be wisely chosen precautions and have aided tremendously in practical programming. The additional burden of type checking by the compiler has been much more than compensated by the amount of confidence gained in the final programs. And this is what good language design should mainly aim for.

COMBINING FEATURES INTO A LANGUAGE

A characteristic of a well-designed feature is that is does not imply any unexpected, hidden inefficiencies of implementation. The packed record structure shown above displays this property: a compiler has full knowledge of the address of each such variable and of the position of the components within a word. It can therefore generate appropriate and efficient instructions for access, packing and unpacking. The whole advantage of this scheme, however, immediately vanishes, if, for example, we introduce so-called dynamic arrays, that is, if we allow information about the actual dimensions of an array to be withheld from the compiler. The textual scan of the program does not reveal the amount of storage needed; as a consequence dynamic allocation must be used involving indirect addressing. This not only impairs the efficiency of the code, but - more importantly - destroys the whole scheme of storage economy.

This is but one example for many that could be listed to show how the combination of two seemingly harmless and well-understood features may suddenly have disastrous effects. A capable language designer must not only be able to select appropriate features, but must also be able to foresee all effects of their being used in combination.

Naturally, one might suggest that a compiler be designed that generates efficient and dense code when the component sizes are known, and less effective code otherwise. But this attitude leads to the optimising monster compilers so well known for their bulkiness and unreliability. Even more significant is the consideration that a good language should not only aid the programmer in avoiding mistakes, but that it must also give him an idea of the complexity and effectiveness of the features it offers. However, if the use of the same feature under only slightly different circumstances yields widely different factors of economy, then the language clearly lacks this highly desirable property. It is very important that the basic method of implementation of each feature can be explained independently from all other features in a manner sufficiently precise to give the programmer a good estimate of the computational effort involved. Some modern languages fail miserably when measured on this criterion.

Transparence is particularly vital with respect to storage allocation and access technique, since storage access is such a frequent operation that any unanticipated, hidden complexity can have disastrous effects of the performance on a whole program. In fact, I found that a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details". Transparence of access mechanism can be achieved by neatly categorising data structuring facilities with respect to applicable access technique. This rule was taken as a guiding principle in the design of the language Pascal, which offers the following structuring facilities:

1. Arrays. Components are selected by computable index. Their offset calculation must in general be deferred until execution time (using index registers if available).

2. Records. Components are selected by a fixed selector name. Their address can therefore be evaluated entirely at compile time. As the compiler may retain their offsets individually in a table, the components are not restricted to be of the same size and type, as in the case of arrays.

3. Sets. Components are not individually selectable at all. Instead, the membership operator in allows to test for their presence or absence. If the size of sets is sufficiently small, they can be represented by their characteristic function fitting into a single word.

4. Files (sequences). Since the length of a sequence may vary during program execution, a dynamic allocation mechanism is required. But it is considerably simplified, because only sequential access is permitted through a "window" displaying the component at the current position.

Variables of these fundamental structures can either be declared explicitly, or they may be invoked dynamically. In the first case they are known by their identifier, in the latter they must be accessed via pointer.

Knowledge of these access characteristics is vital for the programmer, as it is indispensible for the selection of data representation suitable for the algorithm. Regrettably, the current trend in language design seems to move in the opposite direction, namely to obscure these differences. The common excuse is that through the development of more suitable and more efficient hardware these differences would gradually disappear. The fact remains, however, that supposedly more suitable hardware becomes phenomenally complex. It is no longer economical to realise it directly in terms of electronic circuitry, and a new technique has therefore been invented - microprogramming. The essence of this development is that complicated features become the standard with their weird complexity well disguised, and that the programmer is denied the possibility to solve his tasks by simpler means. He doesn't even have a possibility to measure the built-in inefficiencies through quantitative comparisons!

LANGUAGE DESIGN IS DECISION MAKING

From the foregoing it may appear that the secret of good language design lies in a few rules and a sound attitude. In pracitce, of course, the designer is confronted with a bewildering variety of demands from various agents ranging from theoreticians to practitioners, from novices to experts, from revolutionaries crying for innovations to conservatives emphasising compatibility. Let me list a few of the most frequently encountered demands.

- The language must be easy to learn and easy to use.
- It must be safe from misinterpretation and misuse.
- It must be extensible without change of existing features.
- There must be a rigorous, mathematical definition, based on axioms and withstanding the scrutiny of logicians.
- The notation must be convenient and compatible with widely used (and sometimes not so logical) standards.
- The definition must be machine independent, that is without reference to a particular mechanism.
- The language must allow to make efficient use of the facilities of the available computer.
- The compiler must be able to generate efficient code and economise storage.
- The compiler must be fast and compact; it should be void of complex optimisation routines that are rarely used.
- The definition must be self-contained and complete. A reader must easily be able to identify the facilities needed for his purpose.
- The implementation must provide ready access to other facilities available on the system, such as program libraries and (therefore) subprograms written in different languages.
- The language and its compiler must be easily adaptable to different environments with different character sets and different operating system facilities.
- The compiler must be easily portable to other computers. Almost all of it should be conceived without specific reliance on a given order code and storage organisation.
- Time and cost for developing compiler and documentation must be minimal.

It is plain that several of these points are contradictory, but certainly not all of them. The designer must decide where he wishes to place emphasis. It is his task to find a carefully balanced compromise. In fact, the reconciliation of conflicting demands by well chosen compromises is an essential part of every engineering profession; language design should therefore be regarded as a typical engineering discipline. It can be mastered only by experience, and experience is usually gained only after a few failures! The designer's task is even aggravated by the fact that the conflicting demands come from different people whom he is supposed to serve. Whatever he decides, he should never expect unanimous approval.

However, I do not wish to convey the impression that sytisfying one criterion must necessarily mean sacrificing another. True progress appears through the invention of facilities that cater to several seemingly contradictory aims. The three examples mentioned before demonstrate that such progress is indeed feasible.

LANGUAGE DESIGN IS COMPILER CONSTRUCTION

In practice, a programming language is as good as its compiler(s).The believe that it should first be designed entirely in the abstract realm of, say, a set of axioms or an official document, is equally mistaken as the opinion that it must grow out of a practical experiment of implementation before being neatly documented. A successful language must grow out of clear ideas of design goals and of simultaneous attempts to define it in terms of abstract structures, and to implement it on a computer, or preferrably even on several computers. It

follows that experience in compiler construction is a prerequisite to successful language design. Compiler design courses have indeed appeared in the curricula of many computer science department, and seem to be regarded as the epitome of the software craft. Unfortunately they are often strongly biased toward the aspect of syntax analysis, since much theoretical work has been done in this subject. However, the deep penetration of this branch of theory has been of rather small benefit to language design and sometimes was even detrimental. I am afraid that it has misled many language designers to believe that the complexity of a language's syntax was of no concern, since an appropriate parsing algorithm, if not already available, could readily be found for any construction introduced. But it is evident that a language that is simple to parse for the compiler, is also simple to parse for the human programmer, and that can only be an asset. Moreover, the real challenge in compiling is not the detection of correct sentential forms, but coping with ill-formed, erroneous programs, in diagnosing the mistakes and in being able to proceed in a sensible way.

The really essential prerequisite for successful compiler construction is experience in the development of large, complex programs. This includes mastery of techniques in structuring programs and data in general, and in selecting methods for various tasks in particular, such as for scanning of text, construction and search of symbol tables, and composition of code sequences.

One is inclined to wonder where the training of so many compiler and language designers will lead, and whether it is justified. Here I should like to point out that a general appreciation of compiler principles will help the understanding of computer operations and promote the state of the art of programming at large. But there is no reason to believe that the growth rate of the population of programming languages is thereby going to decrease. The emphasis in new developments, however, is gradually shifting from general purpose toward application oriented languages. It is precisely toward this trend that design courses should be directed: exposition of features and presentation of techniques that are common to most areas amenable to algorithmic solution. Such features are, for instance, the fundamental control concepts of sequencing, conditioning, selection, repetition, and recursion. They form a well established basis from which a designer can proceed to fill the given framework with specific facilities oriented toward his particular task and area of application (11).

CONCLUSIONS

I have tried to convey a picture of the problems, challenges, and ordeals facing a language designer, and to draw some lessons from experience gained in the design of a series of languages and compilers. To conclude, let me summarise these lessons learned.

- If you wish to develop a language, you must have a clear idea of how it is intended to be used.
- Keep in mind that a programming language is of no use without an efficient, reliable compiler and a clear, readable documentation. This should provide sufficient incentive to keep the language as simple as ever possible.
- Do not equate simplicity with lack of structure or limitless generality, but rather with transparence, clarity of purpose, and integrity of concepts.
- Adhere to a syntactic structure that can be analysed by simple techniques such as recursive descent with one-symbol lookahead. This not only aids a compiler, but also the programmer, and is vital for successful diagnosis of errors.
- Identify the basic abstractions on which the language is to be based. Try to define the language in terms of a mathematical formalism. This may help to detect hidden inconsistencies and to eliminate notions that cannot be understood in terms of the given abstractions.
- Do not consider the establishment of a formal definition as an end in itself. In particular, the formal definition cannot be a substitute for an informal presentation and for tutorial material. It is mostly an aid to the designer but not a user's document, in which mathematical rigor will contribute to volume but seldom serves the programmer's needs.
- Choose the basic features from the same level of abstraction. Obtain a clear idea on how to represent them in terms of a computer's order code and store. Be aware of the consequences arising from the coexistence of all the various features. They can sometimes be surprising and disastrous.
- Do not hesitate to exclude certain features that prove to be incompatible and too costly in terms of implementation. The fact that other languages include them is no guarantee for their indispensability.
- Obtain a sketch of the complete language before starting work on the compiler. Refrain from adopting highly controversial features; language changes are usually costly in time and effort even during development, and are virtually impossible after a compiler's release, if the language is successful.
- Design the language such that most checking operations can be performed at compile time and need not be deferred until execution. The concept of static data types of variables is essential in this respect, and enhances both programming security and system efficiency.
- Keep the responsability for the design of the language (and possible changes) confined to a single person. If implementation work is delegated, keep closely in touch with it, and make sure to obtain adequate feedback. Beware of programmers who will quietly find solutions no matter what they cost.

And finally, when the project is at its end, carefully reassess it, recognise that many aspects could be improved, and do it all over again.

REFERENCES

[1] T.E. Cheatham, Jr., The recent evolution of programming languages, Information Processing 71 (ed. C.V. Freimann), North-Holland Publ. Co., Amsterdam, 1972, 298-313.

[2] J. Sammet, Programming languages: history and fundamentals, Prentice-Hall, Englewood-Cliffs, 1969.

[3] P. Naur, Programming languages - Status and trends, Proc. NordDATA 72, Helsinki 1972, 36-38.

[4] H.D. Huskey, R. Love, N. Wirth, A syntactic description of BC NELIAC, Comm. ACM vol. 6, no. 7, 367-375 (July 1963).

[5] A. van Wijngaarden, Generalised ALGOL, Ann. Rev. in Autom. Programming 3, (1963) 17-26.

[6] N. Wirth, H. Weber, EULER, A generalization of ALGOL, and its formal description, Comm. ACM vol. 9, no. 1 and 2, 13-23, 89-99, and no. 12, 878 (Jan., Feb., Dec. 1966).

[7] C.A.R. Hoare, An axiomatic basis for computer programming, Comm. ACM vol. 12, no. 10, (Oct. 1969) 576-581.

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

[9] C.A.R. Hoare, Hints on programming language design, SIGACT/SIGPLAN Symposium on priciples of programming languages, Boston, Oct. 1973.

[10] N. Wirth, The programming language Pascal, Acta Informatica vol. 1, (1971) 35-63.

[11] M.V. Wilkes, The outer and inner syntax of a programming language, Comp. J. vol. 11, no. 3, (Nov. 1968) 260-263.