SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1, 309–333 (1971)

The Design of a PASCAL Compiler

N. WIRTH*
Eidgenössische Technische Hochschule, Zürich, Switzerland

SUMMARY

The development of a compiler for the programming language PASCAL[1] is described in some detail. Design decisions concerning the layout of program and data, the organization of the compiler including its syntax analyser, and the over-all approach to the project are discussed. The compiler is written in its own language and was implemented for the CDC 6000 computer family. The reader is expected to be familiar with Reference 1.

KEY WORDS Programming language Compiler design

INTRODUCTION

The programming language PASCAL[1] was designed and implemented with the following principal aims:

(1) To make available a notation in which the fundamental concepts and structures of programming are expressible in a systematic, precise and appropriate way.
(2) To make a notation available which takes into account the various new insights concerning systematic methods of program development.[2,3]
(3) To demonstrate that a language with a rich set of flexible data and program structuring facilities can be implemented by an efficient and moderately sized compiler.
(4) To demonstrate that the use of a machine-independent language with flexible data and program structures for the description of a compiler leads to an increase of its readability, verifiability and consequently its reliability, and that this gain need not be offset by any loss in efficiency.
(5) To gain more insight into the methods of organizing large programs and managing software projects.
(6) To obtain a 'home-made' tool which can (due to its modularity and systematic structure) easily be adapted to various needs, such as, for example, a compiler system effectively usable for class teaching with a large set of error-checking facilities, or a system designed to gather a wide range of user statistics.

Although these points are strongly interrelated, this paper concentrates on the design of a *compiler* for the given language PASCAL, neglecting the fact that the developments of language and compiler were interdependent. Point (4) in particular required that the compiler was to be written in its own language. As a consequence, the implementation had to consist not only of an efficient compiler, but primarily a compiler generating efficient code. To show that these requirements were compatible with the desire for a comprehensive set of structuring facilities in the language was the major challenge of the project.

This paper discusses first the representation of PASCAL programs as machine code, the layout of programs and data at run-time. Then follows a presentation of some aspects of

* Now at Computer Science Department, Stanford University, California 94305, U.S.A.

Received 2 July 1971
Revised 15 July 1971

the compiler, in particular the descriptions of objects during compilation, and the chosen method of syntax analysis. The next section gives an account of the over-all approach to the development of the compiler which is based on the well-known bootstrap technique.

Naturally, a project of this size cannot be described fully in a short paper, and need not be. Many of its problems and techniques are widely known, and their solutions and improvements are a matter of engineering know-how and discipline rather than of scientific principle. The paper therefore concentrates on aspects which are either considered to be novel or important for the understanding of the entire compiler organization or both.

DATA AND PROGRAM LAYOUT

The primary unit of program structure in PASCAL is the procedure. Each procedure (or function) is represented by a data segment and a program segment. The *data segment* contains all variable data local to the procedure, and the *program segment* contains the invariant executable code and the constants. The segments are disjoint. Recursive activations of procedures can therefore use the same program segment; as a consequence, program segments can be allocated 'statically' by the compiler. Data segments, on the other hand, are allocated at each activation of a procedure and released upon exit. Allocation can occur in the fashion of a *stack*.

Data segments and addressing of variables

Since data segments allocated in the stack may have different lengths, they are linked by pointers (addresses) located in the *header* of the data segment. The pointers form two chains, one of them connecting all existing data segments in the order of their generation, the other linking only those data segments which are accessible from the currently executed program segment. The former is called the *dynamic link* (chaining segments in the order of their dynamic generation), the latter is called the *static link* (chaining segments in the order of their static nesting in the program).[4,5] In addition to the two link pointers, the header contains the 'frozen' program status (address) whenever this program calls a procedure. The header is described as

```
type header = record
                slink, dlink: ↑stack;
                pstatus: address
              end
```

The location of a variable is determined by the compiler as an address relative to the origin of the data segment in which the variable is contained. Absolute addresses can be computed upon access to a variable only when the origin of the specific segment is known, and are obtained by addition of the displacement to the origin.

$$a_{abs} = org_{seg} + a_{rel}$$

This deferring of the calculation of variable locations is necessary, because (in the case of recursive activation of procedures) several copies of the same data segment may be existing at one time, and it has the further advantage that at any time storage is only reserved for those variables which are actually in use. The stack mechanism results in an automatic overlay of data belonging to non-concurrently activated procedures.

The most crucial single issue in the realization of an efficient system is the calculation of variable addresses. The efficiency of a system stands or falls with the efficiency of the address calculations. It is obviously necessary that full use be made of available hardware in

this respect. In a computer which contains a facility for indexing addresses, the obvious solution is to load the index register with the relevant segment origin. This nevertheless requires two storage references for every variable access, a fact which has often been mentioned as an argument against the facility of recursion making this double reference necessary. If, however, a computer is available with a *set of index registers*, the best solution is to reserve the index registers for storing the segment origins permanently (i.e. during the entire lifetime of each segment), thus resulting in only one storage reference per variable access. Since index registers are usually available only in small numbers, a practical solution lies in keeping only the origin address in the registers belonging to currently referenceable segments (or even only to the most recently allocated segment). This requires that upon each entry (and possibly also upon exit) of a procedure one (or more) registers must be assigned a new origin address. By a suitable arrangement of the segment links, this 'overhead' can be kept to a negligible minimum well worth expending (cf. following subsection). The set of registers containing the origin addresses of the reference segments forms an array where the ith element contains the origin of the data segment with static *nesting level* i and is called the *display*.[4] The PASCAL 6000 implementation reserves five registers for this purpose, and thus allows for a static nesting depth of procedures up to five. Given an address generated by the compiler in the form of a pair $\langle level, displacement \rangle$, the computation of the absolute address is formulated as

$$a_{abs} := display[level] + displacement$$

and can be executed by a single indexed instruction.

Program segments

Program segments consist of the following sections stored consecutively:

entry code
procedure body code
exit code
constants

The entry and the exit codes are intimately dependent on the data segment allocation scheme, the usage of a 'display' register set and the usage of hardware registers. The entire sequence of instructions executed upon call of and exit from a procedure is listed below. The following variables are used:

P	program status register
D	display
S	pointer to the top of the stack
T	pointer to the most recently generated program segment
W	temporary variable

(In the PASCAL 6000 system D, S, T are held in the B-address registers, W in a fixed work register and P is the program address register.)

Procedure call:

	$W := P$		
	$P := x$		jump
x:	$T\uparrow.pstatus$	$:= W$	store program status
	$S\uparrow.dlink$	$:= T$	extend dynamic link chain
	$S\uparrow.slink$	$:= D_{n-1}$	extend static link chain

$$
\begin{array}{lll}
T & := S & \\
S & := S+s & s = \text{size of new data segment} \\
D_n & := T & \text{define new display entry}
\end{array}
$$

The first instruction group is the code for calling a procedure, the second and the third form the procedure entry code. The second group defines the link values of the header of the new data segment, and the third group defines the new values of the S and T pointers and the display entry corresponding to the new segment.

Procedure exit:

$$
\begin{array}{ll}
S := T & \text{reset stack pointer} \\
T := T\uparrow.dlink & \text{reset segment link pointer} \\
P := T\uparrow.pstatus & \text{reset program status (return jump)}
\end{array}
$$

$$
\left.
\begin{array}{l}
D_m := T \\
D_{m-1} := D_m\uparrow.slink \\
\quad \ldots \\
D_n := D_{n+1}\uparrow.slink
\end{array}
\right\} \text{update display}
$$

The first group of instructions forms the procedure exit code, the second is located in the calling program immediately following the procedure call code and effectuates the resetting of the display to the state before the call. Updating is performed by copying the static link chain into the display; the number of required update operations is $m-n+1$, if $n \leqslant m$, and 0 otherwise, where

m = level of calling procedure
n = level of called procedure

Example:

```
procedure P; begin ... end;
procedure Q;
  procedure R; begin ... P ... end;
  begin ... R ... end;
begin {main program} ... Q ... end;
```

The states of the chains and pointers before and after activation of P are illustrated in Figure 1.

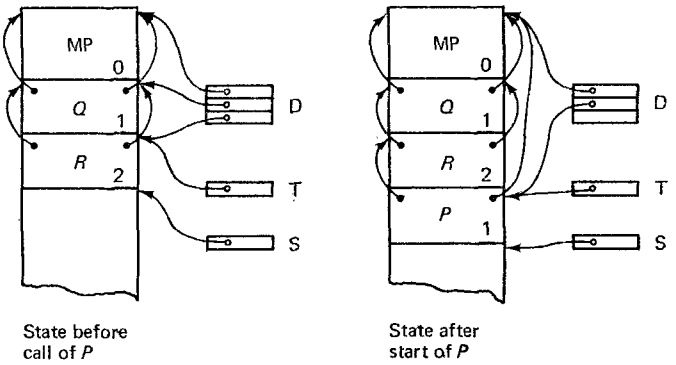


Figure 1. Linkage of data segments

The representation of data

An important decision in the design of PASCAL was the adoption of the rule that all variables within one data segment are to be allocatable by the compiler. As a consequence,

the dynamic array feature of ALGOL 60 was eliminated. The advantage lies mainly in the possibility to compute all addresses at compile-time and to optimize certain index calculations due to the knowledge of array dimensions (cf. section *Code optimization*). The problem of storage allocation and data representation consists in using the program's *type definitions* to design mappings of the abstract data components onto the storage structure of the computer which usually is a homogeneous, one-dimensional array of *physical storage units* (PSU), often called 'words'. The choice of these mappings will depend somewhat on the available PSU size; they are here described as applied to the CDC 6000 computer with a 'wordlength' of 60 bits. This computer is typical insofar as access to a part of a PSU is definitely more time and code consuming than access to an entire PSU. As a consequence, entire variables always are assigned at least one full PSU (no packing).

Scalar types

The symbolic constants $c_0, \ldots, c_n$ of a scalar type are represented by the integers $0, \ldots, n$.

Array types

All components are allocated an integral number of PSU's (size = s). If l is the lower index bound and a is the address of an array variable, the address w of its component with index i is computed as

$$w = a + (i - l) * s = (a - l * s) + i * s = a' + i * s$$

It is to be noted that the first term is computed by the compiler. Multi-dimensional arrays are considered as arrays with array-structured components. This leads to conventional polynomial index calculations.

Record types

Since all record field types are known to the compiler, the field displacements can be determined, and as a consequence field designators can be evaluated to a fixed address and require no run-time indexing. Every field normally occupies an integral number of PSU's.

The syntactic rules imply that a possible variant part always must follow the fixed part. This is a necessary condition to keep the displacement addresses of the fields of the fixed part independent of the chosen variant. If a component of an array or a record has itself a record structure with a variant part, its size is chosen as the size of the largest possible variant. If a record type is defined as *packed* (not described in Reference 1), then fields of scalar, pointer or powerset type are assigned to partial PSU's. However, each packed field will always reside within only one PSU. The amount of storage needed in bits is

$$[log(max(abs(m), abs(n)) + 1)] + 1$$

for scalar types defined as subrange $m..n$,

$$[log\ (n)] + 1$$

for scalar types defined as $(c_0, c_1, \ldots, c_{n-1})$, and

$$n + 2$$

for powerset types defined as **powerset** $m..n$ $(0 \leqslant m \leqslant n)$. (The extra bit is introduced because it facilitates field extraction with the available shift instructions of the CDC computer.)

Example:

```
x: packed record a, b, c: 0 .. 999;
                 d, e: char;
                 f, g: boolean;
                 h: powerset 0 .. 7
   end
```

The variable x will be represented in one 60 bit PSU as follows:

x	a	b	c	d	e	f	g	h
	11	11	11	7	7	2	2	9

It was considered essential that the choice of whether a record is to be represented in a packed or unpacked form should be left to the programmer. Although access to a field in a packed record (particularly assignment) is more time consuming, the use of packed records for the compiler's own object table caused no significant increase of compiling time (2 per cent), but allowed for a considerable reduction of storage needed (55 per cent for the object table, 15 per cent for all other data).

Powerset types

A very efficient implementation of powerset operations is possible, if a powerset value can be accommodated within one PSU. The presence of element c_i in a powerset value is then representable by bit i. The set operations of PASCAL are realized by the conventional bit-parallel logical instructions ('and' for intersection, 'or' for union), and the test for membership (relation **in**) is realized by shifting the appropriate bit into a testable position (e.g. the sign bit position).

The restriction of allowing powersets to be constructed only upon base sets whose number of elements is not larger than the PSU length of the computer guarantees an efficient and economic implementation of set operations and powerset variables, which constitute a practically machine independent abstraction of the structure commonly known as the bitstring.[6]

File types

As the only sequential access structure in PASCAL, the file is particularly suited for allocation on mechanically moving storage devices. Since usually the administration of secondary storage is handled by the computer's operating system, an implementation of file variables involves interfacing problems with the existing operating system. The CDC SCOPE Operating System fortunately offers simple and efficient interfacing mechanisms and, as a consequence, all PASCAL file variables are assigned to secondary storage.[8]

The simple and clearcut interfacing between the PASCAL program and the SCOPE system is predetermined by the configuration of the hardware which is divided into central and peripheral processing units (CP and PP's), and deserves mentioning although it does not involve any basically novel features.

To each file variable is allocated a file descriptor and a fixed-size file buffer in main store (by PASCAL), and a (variable size) file space in secondary store (disk, tape or extended core store, by SCOPE).

The data transfer between the main store buffer and the secondary store is performed by a Peripheral Processor (PP, channel) programmed by SCOPE. The buffer always contains

(at least) the component denoted by the PASCAL file pointer. As a consequence, that element is directly accessible at the location to or from which it is transferred from or to secondary storage by the PP, and no further costly in-core data transfer is necessary like in practically all other common operating systems. The CPU actions caused by the standard procedures *put* and *get* merely change pointers. This fact considerably increases the efficiency of I/O-bound programs.[7]

The size s of the buffer is determined by the compiler according to two considerations:

(1) Since the entire current file element $f\uparrow$ is to be directly accessible within the buffer, s must be a multiple of the file component size s'. ($s = n*s'$.)

(2) Since the file is represented on secondary store by a sequence of so-called physical record units (PRU's) with fixed length l, and since a data transfer can only be performed with entire PRU's, the buffer must be able to hold at least one PRU ($s > l$).

Most often it is advisable to choose a buffer capable of holding several PRU's on a machine with a large main store, in order to reduce the number of channel activations.

The file descriptor contains the address and size of the buffer, and the pointers necessary to operate on the buffer in the usual circular mode[8] [see section *The description of objects during compilation*].

Every file component is allocated an integral number of PSU's. Only character files represent an exceptional case insofar as the SCOPE system uses a packed representation for them (10 characters per PSU). With respect to the possibility of writing machine independent programs, the notion of a character file is often of great importance, and is therefore implemented with an additional implicit routine for packing or unpacking. This is the only instance in the PASCAL system where data are packed or unpacked without the explicit control or even awareness of the programmer.

Class types

Pointer variables hold as their value an absolute address of a component of the class variable to which they are bound and which is called their *domain*. Class variables are allocated an area of main store whose size is calculated by the compiler from the size of the class component and the maximum number of compenents indicated in the class type definition. In addition, with every class variable is associated an allocation pointer which is initialized to the origin of the class area, and which is augmented by the size of a requested component upon each call of the standard procedure

alloc(p)

If the components of the domain of p have a variant record structure, the size is determined by the additional parameter designating the particular variant desired for the new component. Classes may thus consist of components of different sizes, and allocation takes place with the greatest possible storage economy.

The pointer value **nil** is represented by an address value beyond the physical addressing space. This forces an addressing error trap upon inadvertant reference to a component via a pointer variable with value **nil**, and obviates further (expensive) checking by means of software. There is no automatic garbage collection.

Procedure parameters

There exist three kinds of procedure parameters:

(1) Parameters denoting a constant.

(2) Parameters denoting a variable.

(3) Parameters denoting a procedure or a function.

This scheme offers the same set of object kinds for formal quantities as are available for actual quantities, with the exception of types which cannot be passed as parameters. A specific section of the data segment is used to store representations of the parameters.

In the case of a constant parameter, the value itself is stored in the parameter location, if it is 'a small quantity', i.e. fits into one PSU. Otherwise, the address of the actual parameter is stored as formal parameter representation, and references to it will be indirect. Thereby copying of large objects is avoided. In contrast to ALGOL 60's value parameter, the definition of the constant parameter leaves the choice of parameter passing technique open to the compiler. In any case, no assignment to a constant parameter will be allowed, and violations against this rule can be detected by the compiler; situations analogous to ALGOL 60's

procedure *increment*(x); **integer** x; $x := x+1$;
... $p(a+b)$ ...

cannot occur.

A variable parameter will always be passed as the address of the actual variable evaluated at the time of calling the procedure.

In order to represent a procedure uniquely, two components are necessary: the address of the entry point of the code and the address of the data segment of that procedure in which the parametric procedure is declared locally.

Code optimization

The language was designed so that implementation of most of its constructs should be possible without a need for extensive optimization processes on conventional computers. The compiler described here restricts optimizations to strictly context-free situations. Thus it does not contain any context-dependent optimization capabilities such as detection of common sub-expressions (e.g. $a[i+j] := a[i+j]+1$), linear index progression (in matrix computation), or extraction of constant parts from loops (e.g. **for** $k := 1$ **to** n **do** $b[i,j] := a[i,k]*a[k,j]+b[i,j]$, where the address of $b[i,j]$ need to be computed only once). The most important single feature is that computations on known values are executed by the compiler whenever possible; this concerns particularly address calculations as exemplified by the following case:

Given the variable

```
x: array [10 .. 25] of
   record a: integer;
          b: array [-13 .. 0] of
             record u: char; v: real
             end;
          c: boolean
   end
```

all of the following variable designations will be represented in the generated code by one evaluated address:

x
$x[13]$
$x[15].a$
$x[19].b[-7]$
$x[25].b[-5].v$

A second important optimization concerns the implementation of the arithmetic operations of integer multiplication and division when one operand is a constant. This optimization is particularly appropriate for the CDC 6000 computer which, on the one hand, has no single instruction for these operations, but where, on the other hand, shift operations and additions are fast. Thus, the compiler uses a shift instruction in the case

$$x \textbf{ div } c \quad \text{if } c = 2^n \quad (n \geqslant 0)$$

and a sequence of shifts and additions in the case

$$x*c \quad (\text{or } c*x) \quad \text{if } c = 2^n \quad \text{or} \quad c = 2^m + 2^n \quad \text{or} \quad c = 2^m - 2^n \quad (m > n \geqslant 0)$$

These optimizations are, of course, also employed in index computations. Given, as an example, a variable

$$a\textbf{: array } [1 \,..\, 10, 1 \,..\, 20] \textbf{ of } \textit{real}$$

the computation of the address of the component variable $a[i+1, j+1]$ results in the generation of only two fetch instructions (for i and j), two shift and two add instructions. At run-time, the value $i*16+i*4+j$ will be used as index to the address of the virtual variable $a[0, 0]$.

THE COMPILER

General organization

The key to an efficient compiler for a fast computer with relatively large main store is the *one-pass scheme*. It minimizes the number of references to secondary store which involve the operating system and exceed all other processes by orders of magnitude of time-consumption. The restrictions imposed on the language due to the choice of a one-pass compiler are minimal (objects have usually to be declared textually prior to being referenced) and the complications due to unavoidable forward references are small.

The PASCAL 6000 compiler generates code into an area of main store. As soon as a procedure has been compiled, its code is transferred to a file on secondary storage. Since procedure declarations must be placed ahead of the statement part, there is at any time only code belonging to a single procedure in main store. The code buffer therefore has to be no larger than is necessary to keep the entire code of the longest procedure (usually the main program).

The compiler generates *absolute code*. The gain in compilation speed and particularly in avoiding the use of standard relocation loaders more than outweighs the doubtful advantage of being able to merge 'binary' programs after compilation. If a compiler is available with a speed practically equal to that of a relocation loader or linkage editor, *there is no point in keeping relocatable binary* versions of a program; merging of programs should occur at the source language level. The first software system (known to the author) based on this premise was Burroughs' B5500 Extended ALGOL.[9] Although the scheme was highly successful, it has found few followers, partly because manufactures refuse to produce faster compilers for standard languages, partly because new languages with highly intricate features force compilers to remain slow and dependent on costly optimization algorithms.

The description of objects during compilation

All identifiers occurring in a program are stored in a table along with a description of the object they name. Since every object is characterized by various attributes with different

ranges of values, the record is the appropriate data structure. Since various objects are described by different sets of attributes, the record has a variant part. The table itself is structured as a class of such records, allowing a compact storage representation and fast access to components through pointers. The declaration of the object table is given below; it uses the following data-types:

```
type pointer   = ↑objecttable;
     shortint  = -2^17 + 1 .. 2^17 - 1;
     address   = 0 .. 2^16 - 1;
     level     = 0 .. 5;
     bitrange  = 0 .. 59;
     idclass   = (type, constant, variable, procedure, field,
                  tagfield);
     typeform  = (numeric, symbolic, pointer, powerset, array,
                  record, class, file);
     idkind    = (actual, formal);

var objecttable: class of
  packed record name: alfa; {identifier}
    nextelement: pointer; {link to next object in same scope}
    case class: idclass of
    type: (size: shortint;
           case form: typeform of
           numeric: (bitsize: bitrange;
                     min, max: integer);
           symbolic: (firstconstant, pwset: pointer;
                      bitsize 1: bitrange);
           pointer: (domain, elementtype: pointer);
           powerset: (basetype: pointer);
           array: (arrayelementtype, indextype: pointer;
                   lowbound, highbound: shortint);
           record: (firstfield, recvar: pointer);
           class: (classelementtype: pointer);
           file: (fileelementtype: pointer));
    constant: (constanttype: pointer;
      case constantkind: idkind of
      actual: (successor: pointer;
               value: integer);
      formal: (constantaddress: address;
               constlevel: level));
    variable: (variabletype: pointer;
               variablekind: idkind;
               variableaddress: address;
               variablelevel: level);
    procedure: (proceduretype, parameter, context: pointer;
                procedurekind: idkind;
                procedureaddress: address;
                procedurelevel: level;
                datasegmentsize: shortint);
```

```
        field: (fieldtype: pointer;
                fieldaddress: address;
                displacement, width: bitrange);
        tagfield: (casesize: shortint;
                   firstvariant: pointer;
                   case tagvalue: boolean of
                   false: (casetype: pointer);
                   true: (casevalue: integer))
    end
```

The object table contains descriptions of all named objects. Anonymous objects, which are generated during compilation and correspond to component variable denotations, factors, terms, expressions, etc. are described by variables local to the various processing procedures.

Their type, called 'attribute', is specified as follows:

```
type attributekind =
        (var, shortvalue, loadedvalue, loadedcondition);
attribute =
  record type: pointer;
    case kind: attributekind of
    var: (access: (direct, indirect, indexed);
          base: level;
          displacement: shortint;
          case packed: boolean of
            false:;
            true: (bitaddress, bitsize: bitrange));
    shortvalue: (value: shortint);
    loadedvalue: (constantterm: integer);
    loadedcondition: (jump: 0 .. 3;
                      arithmetic: boolean)
  end
```

(If an anonymous object is either a loaded value or a loaded condition, the number of the register into which it is loaded is implicitly known, because the available work registers are treated as a stack. It is therefore not necessary to include this number in the attribute record, although it is an essential item of the object's description.)

The complete data type definitions used by the compiler to describe objects are here included not only to convey an insight into the compiler's organization, but also to demonstrate the power of PASCAL's data definition facilities. They allow for a transparent and fully symbolic, machine-independent form of data specifications, but at the same time make an economic usage of storage possible. The combination of these two characteristics—availability of systematic and appropriate abstractions, and efficient adaptability to real computers—was the principal objective in the design of PASCAL.

Syntactic analysis of the source text

Most programming languages have been developed without due consideration of the influence of their syntactic structure on the complexity and efficiency of parsing methods to analyse them. Considerable research activities on parsing methods were evoked by the challenges issued by ALGOL 60. These activities even reinforced the belief that considerations of syntactic constraints could only impede progress towards more natural and powerful

languages, and would soon become irrelevant with the development of new and clever methods of syntactic analysis.

On the contrary, the guiding idea in the layout of the syntax of PASCAL was simplicity due to the recognition that structures difficult to process by computers are often also difficult to master by human readers and writers, and that the presence of a complicated syntax does not only make more complex and less efficient parsing algorithms necessary, but also increases the likelihood of human misunderstandings and programming errors. It was also recognized that one of the major challenges in developing a processing system for a language is its capability to meaningfully diagnose syntactic errors and to continue processing of subsequent text with a reasonably large probability of correct diagnosis. If the system is to be used successfully in an environment of programming novices, this capability must be assigned no less than highest priority. The problem of syntax analysis thereby obtains entirely new aspects; the compiler must not only process the defined language, but virtually all sequences of symbols of the basic vocabulary. This fact mainly reaffirmed the conviction that the challenge did not lie in designing a 'powerful' language and thereafter a sophisticated parser somehow coping with all constructs of the language, but rather in designing a language based on a syntax allowing the application of a reasonably simple and perspicuous analysis technique.

The chosen method was first described by Conway[10] who called it 'separable transition diagram' technique. The syntax of the language is thereby presented as a finite set of *pseudo-finite-state* recognizers. The attribute 'pseudo' is due to the fact that some of the basic symbols to be recognized are replaced by sentences recognizable by one of the members of this set. The recognizers may thus activate each other, possibly causing recursion. This top-down parsing technique has the following advantages:

(1) Every single recognizer can be presented by a lucid finite graph directly representing the recognizer's program.
(2) If a programming system is available offering recursive procedures, no explicit stack mechanism need be programmed.
(3) Program paths introduced to handle syntactic errors can easily be represented in the syntax graphs and be included in an extended but concise syntax documentation. They can be introduced without constraints due to a given analysis algorithm which possibly uses a tabular representation of the syntax in a rigid format.

Following this technique, the syntax of the language is formulated as a set S of finite graphs each with one entry and one exit point. Each edge connecting two nodes corresponds to a state-transition involving the acceptance of either a basic symbol, if the edge is labelled with that symbol, or of a sentence recognizable by one of the members of S, if the edge is labelled with that member. The translation of diagrams to and from BNF-productions is straightforward, and also the verification of unambiguity of the resulting language poses no sophisticated problems, particularly if no member of S accepts the empty sentence. The two main criteria are that

(a) Each member of S is deterministic, and
(b) The selection of the exit of a diagram is deterministic.

For details the reader is referred to Reference 11. Although these conditions seem rather rigid and restrictive, it turned out to be relatively easy to mould the language according to them. The strict adherence to the constraint of a one-symbol lookahead had actually only two noticeable effects which caused deviations from the ALGOL 60 syntax, namely

(i) The elimination of multiple assignments, and
(ii) The elimination of non-integer labels.

The resulting syntax of PASCAL is illustrated in the Appendix, which reflects with minor deviations directly the organization of the compiler. The dependence relationships between the various main procedures are given by Figure 2. According to common practice, not the individual input characters but rather entities gathered by a scanner routine are considered as basic symbols. The presence of the scanner is not only advantageous with respect to efficiency, but necessary because word-delimiters (such as **begin**) are represented like identifiers (without escape characters) and must be 'intercepted' by the scanner. Identifiers (and also numbers) are therefore considered as basic symbols. It is thus noteworthy, that the *goal-oriented top-down analyser* is obtaining input via a *source-oriented scanner* based on the *bottom-up* parsing principle. These techniques were first used in the Elliott ALGOL translator.[14]

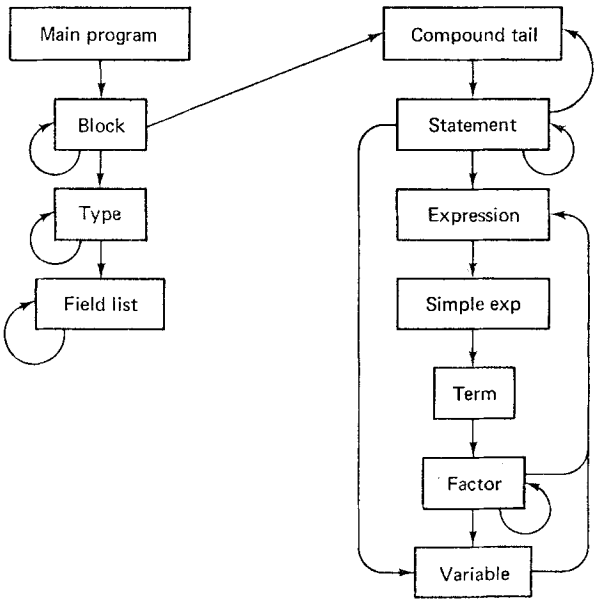


Figure 2. Procedure dependence diagram

Performance and statistical data

Size (approximate data)

The source program:

4000 lines total, containing
130,000 characters (not counting comments)
41% blanks
41% letters,
3% digits
15% special characters
Average length of source line: 33 characters.

The program contains 500 distinct identifiers, and in the average each occurs 18 times. It contains 5000 occurrences of word-delimiters, the most frequent ones being **end, begin, if, then** and **else** with 850, 810, 780, 780 and 400 occurrences respectively.

The object program:

Total field length required:†	19,000 (= $45{,}000_8$) words
Compiler program proper:	67·8%
Object code buffer†	4·7%
Object table†	9·2%
Other data	4·5%
Input and output buffers:	8·3%
Interface and I/O routines:	5·5%

The program consists of 32,700 instructions composed as follows:

(1)

48·7%	long instructions (30-bit)
28·7%	short instructions (15-bit)
22·8%	padding instructions (NOOP)
100%	

(2)

27·6%	fetch and store instructions
15·0%	load literal instructions
3·5%	arithmetic instructions
14·4%	logical and shift instructions
6·2%	base address register instructions
10·5%	jumps and subroutine calls
22·8%	NOOP padding instructions
100%	

(3) The X-registers are used as a stack to hold intermediate results during the evaluation of expressions. Of the 7,927 instructions compiled which assign values to an X-register, the following distribution is obtained:

X1	75·0%
X2	21·6%
X3	3·1%
X4	0·3%
X5	0·02%

(4) The B-registers are used as the display D (cf. section *Program segments*). Of the 1564 instructions compiled which update the display, the following distribution is measured:

B1	84·2%
B2	14·7%
B3	1·1%
B4	not used

The CDC 6000 computers provide transfers of control only to the beginning of words; this explains the large number of padding instructions typical for a program with a high percentage of jumps.

Speed

By measuring the performance on recompilation of the compiler on the CDC 6400 computer, the following data are obtained (use of shorter programs yields better results):

† Adjusted so that compilation of the compiler itself is possible.

Time to load and compile (the compiler source program):

40 sec (CP) + 15 sec (PP)

This yields an average of

100 lines of source code processed per (CP) second,
820 instructions generated per second.

(An expected improvement factor for the CDC 6600 is about 2·5)

COMPILER DESIGN TECHNIQUE

Work on implementation of an earlier version of PASCAL was started in 1968. The compiler was written in FORTRAN with the intention of later being translated into PASCAL and making use of the bootstrapping method. This proved to be a major mistake, and although the compiler was completed, it was decided to start the development of a new compiler from scratch and to use the occasion to modify the language definition in several details. The effect of writing the compiler in FORTRAN, a language obviously unsuited to formulate complicated program and data structures, was that the data definitions (and as a consequence also the program) had to be so twisted to comply to the rules of FORTRAN, that it became easier to write a new compiler than to 'translate' the existing one.

The second compiler, whose development was started in late 1969, was *originally written in PASCAL*. This implied that the language had to be defined already very rigidly, because even minor changes inevitably cause considerable amounts of work to adapt the compiler. After being written in PASCAL, the compiler was then translated 'by hand' into an available low-level language without any attempt at optimization. This process of translation and bringing the compiler to the stage where it was able to compile its own original version required about 1 man-month of effort. At this stage, several features considered unimportant for expressing the compiler were omitted: powerset structures, packed records, functions, procedure- and function-parameters, real arithmetic and all kinds of local code optimizations. Nevertheless, the initial version of the compiler must be considered as relatively large in comparison with the final version which had undergone many bootstrapping stages (> 60 per cent). The program consisted of almost 3,000 lines written without any opportunity of testing—a healthy experience for any software expert. The advantage of starting with a very small initial version lies in the minimal effort of initial hand translation into an available language. This advantage, however, is quite small and must be more than compensated by later efforts to put features to actual use which are introduced in the later bootstrap stages. As witnessed by experience, rewriting the compiler to make use of new features is usually deferred due to other more urgent affairs until the effort becomes so large that it is considered extravagant and not worth undertaking. It is therefore worth while to start out with a set of the language containing all features considered of sufficiently high importance, and whose later introduction would require a considerable rewriting effort. It is noteworthy that in particular the record-, variant record-, file- and class-structures were all present in the initial compiler version.

The compiler was developed by a team of three people with the task divided into the following three parts:

(1) Type definitions, variable declarations and procedure headings including formal parameter lists.
(2) Expressions and statements.
(3) Interface with the operating system.

A result of this partitioning of the task is a minimization of the interrelationships between the parts. The principal interface between parts (1) and (2) is the object table (cf. section *The description of objects during compilation*) which is constructed by the programs of part (1) and inspected by the programs of part (2). The definition of its structure was therefore necessarily one of the earliest and most influential design decisions. A suitable partitioning of the task constitutes one of the more crucial combined design and management decisions in such a project of software engineering, and has decidedly its effect upon the amount and quality of total work expended. In this case, the compiler required roughly two man-years of effort.

The chosen method of development—writing an initial version in the language itself rather than a commonly available one—has its effects upon the choice of techniques to adapt the compiler to other computers. The idea of writing the compiler first in FORTRAN and thereby making it automatically available on practically all computers was one of the motivations leading to the first project (which failed). Although attractive at first sight, it suffers from the fact that in practice FORTRAN versions on different computers are so far from being compatible that the transfer of a large program is still a major undertaking—apart from the fact that the compiler is so obscure that it is very difficult even to make the relatively small changes necessary for the production of a different object code.

The obvious method to transport a PASCAL compiler (not available in any other language than PASCAL itself), is to first rewrite the code generators and their calls to generate code of the target computer (without consideration of optimization). After completion of step 1, the new compiler can be used in step 2 to bootstrap itself, thus generating a binary version for the target computer B on the host computer A which can then easily be transported (see Figure 3). This approach is currently used to transfer the compiler onto an ICL 1900 computer.

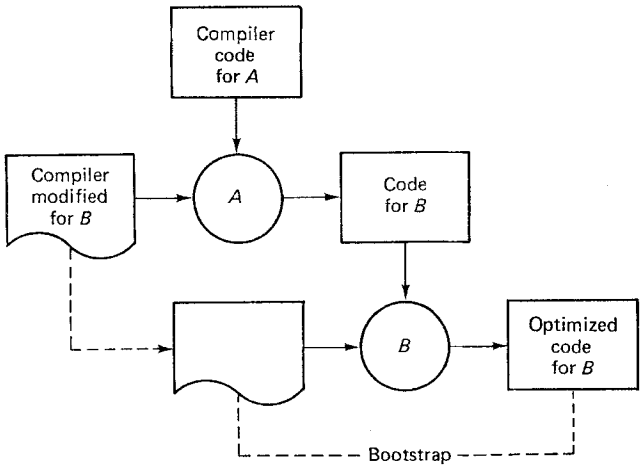


Figure 3. Bootstrap step

If no computer A is available to implementors working on a computer B, then step 2 can be replaced by hand translating the result of step 1 into an equivalent program in a language L available on computer B. The task of this translation is greatly simplified if L offers recursive procedures. The actual efficiency of the available implementation of L has no effect on the final version resulting from the bootstrap process. This approach is currently used to transfer the compiler onto an IBM System/360 computer using PL/1.

ON THE RELATIONSHIP BETWEEN THE COMPLEXITY OF COMPILATION AND COMPUTER ARCHITECTURE

PASCAL is certainly a language designed without any specific computer in mind. The goal was to obtain a language whose programs could be transformed without too great expenditures into codes which would be efficient representations of the original programs for all conventional machines. It seems worth while at this point to investigate which properties of computer architecture are most desirable to keep the necessary expenditures for compiling good code reasonably small, and in particular to measure the CDC 6000 computer by these criteria.

One vital criterion to be able to generate efficient object code is the presence of at least two index (or base) registers. Another, much more general property is an overall simplicity of the instruction set and the entire architecture. The more facilities, such as different sets of registers available, the more complex will be the compilation process making good use of them. A truly good architecture should make so-called optimizations largely unnecessary (or impossible). Recent results in program analysis showed that in most programs the great majority of time is spent in a small percentage of the code, thus making complex over-all optimizations at compile time a rather wasteful expenditure.[12] However, since even selective optimization necessarily requires the presence of complete optimization routines in the compiler, an architecture allowing their elimination is still highly desirable.

The primary virtue of the CDC 6000 architecture is the regularity and brevity of its instruction set. Contrary to popular beliefs, an extensive instruction set does not enhance a computer (at least not to users and designers of compilers) but is rather a symptom of the lack of a unifying, clear concept on the part of the architect. Versatility and variety are more economically provided by software than by hardware, as is demonstrated by the recent upsurge of 'micro-programming' and the results of experiments to compile higher languages directly into 'micro-code'.[13] In spite of the small number of only 64 instructions of the CDC 6000 computer, the compiler uses only 42 of them (66 per cent) in generated code; the percentage figure is expected to be even much smaller for computers with elaborate instruction sets. The number of different instructions used versus the percentage of instructions generated in the compiler's code is represented by Figure 4. It shows that only four instructions account for 64 per cent of the entire code (including the most frequent NOOP).

Considering the fact that the CDC 6000 computer satisfies these two main requirements quite well, its deficiencies appear rather minor. They concern details which, in certain circumstances, may nevertheless be very awkward. The following points have in this sense been noted:

(1) Absence of a jump instruction depositing the current processor status in a general register (RJ's cannot be used in re-entrant code).
(2) Absence of logical shift instructions.
(3) Use of one's complement arithmetic with two representations of zero.

The one major flaw of the CDC 6000 computer is the absence of automatic traps upon overflow conditions on integer arithmetic. Such conditions are simply ignored, and there is no way to detect them. This is even the more aggravating, since some instructions operate on 60-bit integers, some on 48-bit integers, some on 18-bit integers, some extending, some extracting one representation from the other with correct treatment of the sign, but without testing for overflow. This very serious deficiency makes computations using integers rather hazardous. It is to be noted that this 'property' is also present in several other computers,

and that its effects are not restricted to the use of PASCAL. Hopefully, designers of future computers—or of extensions to existing ones—will pay more attention to such seemingly negligible properties.

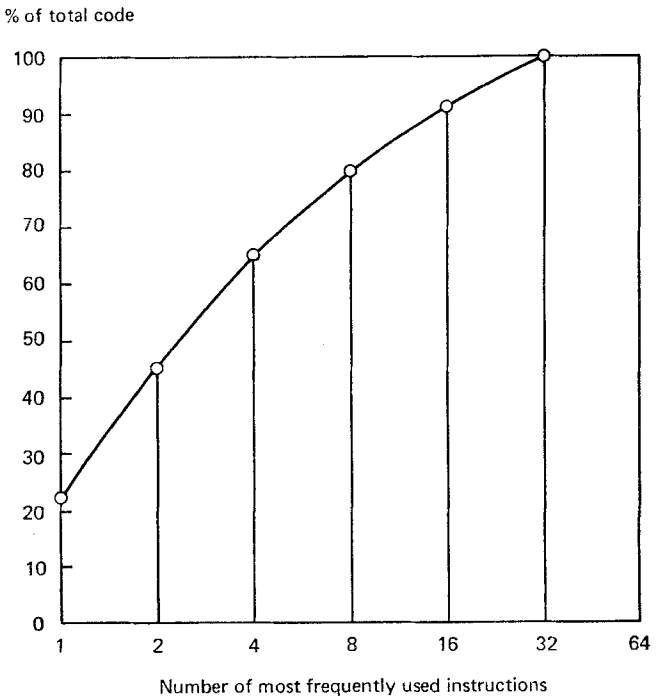


Figure 4. Instruction usage

CONCLUSIONS

Experience with the compiler justifies the conclusion that the principle of designing a sufficiently comprehensive but syntactically simple language, and developing an implementation without extensive optimization features was sound. Its success, however, could easily be inverted by yielding to pressures from various user groups to include extensions according to individual wishes and *ad hoc* desires. The key to a high priority availability of the compiler in a multi-programming system is its 'reasonable' size. Having reached slightly more than one-quarter of core size, caution is recommended against any further extension.

Another design decision that proved to be correct in the light of experience was the limitation of the expression evaluation stack and the display according to the number of available registers. Thus, if a program requires more than five intermediate results in evaluating an expression, or contains nestings of procedure declarations to a depth larger than five, the compiler will produce a message rather than provide a 'software-solution' requiring voluminous compiler routines (rarely used) and implying inefficient code. Into the same category belongs the decision to exclude complicated routines for compiling expressions using a minimal number of registers; they rarely have a chance to do significant optimizations but nevertheless contribute to the expansion of the compiler's size and complexity.[12] It seems that one of the keys to designing a good compiler is having a sense or the intuition of what is important and what is negligible. This engineering intuition is more crucial than extensive knowledge e.g. about the theory of syntax analysis.

The following four programs were written in the three languages ALGOL, FORTRAN and PASCAL and executed on the CDC 6400 computer to allow a rough comparison of the quality of generated code. They were chosen so that an obvious and straightforward translation was possible. Therefore they cannot demonstrate the power of expression of PASCAL; they are limited to the category of programs more or less conveniently expressible in all three languages:

(1) Matrix multiplication $B := A*A$, no output.
(2) Sorting an array of 2,000 numbers.
(3) Finding all possible additive partitions of integers 1–30. (Use of recursion, 'hand-coded' stack in FORTRAN.)
(4) Counting the characters in a file.

For all programs, the total central processor times for compilation (negligible) and execution were measured:

Table I

Program	ALGOL 60		FORTRAN IV		PASCAL	
	sec.	rel.	sec.	rel.	sec.	rel.
Matmult $n = 100$	115·1	5·82	19·8	1	48·5	2·45
					32·1†	1·62
Matmult $n = 64 = 2^6$	32·2	5·85	5·5	1	8·6	1·56
Sort $n = 2000$	150·0	4·38	34·3	1	40·0	1·17
Partition $n = 30$	106·8	8·75	12·2	1	9·28	0·762
Partitition, no output	10·0	6·00	1·66	1	1·20	0·722
Charcount	302·2‡	8·24	36·7	1	7·32	0·20
	360·7‡	9·83				

† Case $n = 96 = 2^6 + 2^5$.
‡ 302·2 using IFIP Standard I/O Procedure *incharacter*, 360·7 using 'ACM-Knuth-I/O' procedure *output*.

Perhaps more important than the system's efficiency is its degree of reliability. This was achieved mainly due to the use of a simple and perspicuous scheme of syntax analysis and of the language PASCAL itself to describe the compilation algorithm and the underlying data structures. As a result, compilation of separate features of the language can be tested separately, and after removing the chiefly clerical mistakes the correctness of the involved routines can be guaranteed with a very high degree of confidence. The choice of the method of 'recursive descent' for syntax analysis is in turn dependent on the availability of a language with recursive procedures, such as PASCAL. In the light of experience, it proved to be a good choice. The inclusion of extensions and the alteration of error diagnostic facilities proved to be easier than with methods using syntax tables, such as the precedence analysis scheme used in the first version of the compiler. An interesting and probably significant side effect of this technique was that the syntax of the language itself was designed in terms of flow diagrams, and not of BNF-productions. These diagrams represent directly the structure of control in the compiler. Each diagram corresponds to a procedure. Since the diagrams have a regular structure, they can all be presented by if-, case-, while- and repeat-statements. Goto statements are used in the compiler only to handle syntactic errors of the source text. This was the most important single fact enhancing the readability and reliability aspects of the compiler program, and of course required a fair amount of discipline on the part of the programmers. Almost equally important, however, was the availability

```
'BEGIN' 'COMMENT' MATRIX MULTIPLICATION :
   'INTEGER' I,J,K,N; 'REAL' S;
   'REAL' 'ARRAY' A,B[1:100, 1:100];
   N := 100;
   'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
   'FOR' J := 1 'STEP' 1 'UNTIL' N 'DO' A[I,J] := 1.0;
   'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
   'FOR' J := 1 'STEP' 1 'UNTIL' N 'DO'
   'BEGIN' S := 0;
      'FOR' K := 1 'STEP' 1 'UNTIL' N 'DO' S := A[I,K]*A[K,J] + S;
      B[I,J] := S
   'END'
'END' :
```

```
      FORTRAN IV PROGRAM MATMULT(OUTPUT)
      DIMENSION A(100,100),B(100,100)
      N = 100
      DO 10 I=1,N
      DO 10 J=1,N
   10 A(I,J) = 1.0
      DO 30 I=1,N
      DO 30 J=1,N
      S = 0
      DO 20 K=1,N
   20 S = A(I,K)*A(K,J) + S
   30 B(I,J) = S
      STOP
      END
```

```
! MATRIX MULTIPLICATION "
   CONST N = 100;
   VAR I,J,K: INTEGER;  S: REAL;
      A, B: ARRAY [1..N, 1..N] OF REAL;
BEGIN FOR I := 1 TO N DO
      FOR J := 1 TO N DO A[I,J] := 1.0;
      FOR I := 1 TO N DO
      FOR J := 1 TO N DO
      BEGIN S := 0;
         FOR K := 1 TO N DO S := A[I,K]*A[K,J] + S;
         B[I,J] := S
      END
END .
```

```
'BEGIN' 'COMMENT' SORTING AN ARRAY;
   'INTEGER' I,J,K,M,N;
   'INTEGER' 'ARRAY' A[1:2000];
   N := 2000;
   'FOR' I := 1 'STEP' 1 'UNTIL' N 'DO'
      A[I] := I;
   'FOR' I := 1 'STEP' 1 'UNTIL' N-1 'DO'
   'BEGIN' K := I; M := A[I];
      'FOR' J := I+1 'STEP' 1 'UNTIL' N 'DO'
      'IF' A[J] > M 'THEN' 'BEGIN' K := J; M := A[J] 'END';
      A[K] := A[I]; A[I] := M;
   'END'
'END' ;
```

```
      FORTRAN IV PROGRAM SORT(OUTPUT)
      INTEGER A(2000)
      N = 2000
      DO 10 I=1,N
   10 A(I) = I
      N1 = N-1
      DO 30 I=1,N1
      K = I
      M = A(I)
      I1 = I+1
      DO 20 J = I1,N
      IF (A(J) .LE. M) GO TO 20
      K = J
      M = A(J)
   20 CONTINUE
      A(K) = A(I)
   30 A(I) = M
      STOP
      END
```

```
! SORT AN ARRAY OF INTEGERS "
   CONST N = 2000;
   VAR I,J,K,M: INTEGER;
      A: ARRAY [1..N] OF INTEGER;
BEGIN FOR I := 1 TO N DO A[I] := I;
      FOR I := 1 TO N-1 DO
      BEGIN K := I; M := A[I];
            FOR J := I+1 TO N DO
            IF A[J] > M THEN BEGIN K := J; M := A[J] END ;
             A[K] := A[I]; A[I] := M
      END
END .
```

```
'BEGIN' 'INTEGER''ARRAY' P[1:100];
  'PROCEDURE' A(M,N,I);
    'VALUE' M,N,I;  'INTEGER' M,N,I;
  'BEGIN' 'INTEGER' Z,K;
    'FOR' Z := 1 'STEP' 1 'UNTIL' M 'DO'
    'BEGIN' P[I] := Z;
      'IF' Z = N 'THEN'
      'BEGIN' OUTPUT(61,'('/,40(BZD)')',(P[K],K:=1:I));
        'GOTO' EX;
      'END';
      A(Z,N-Z,I+1);
    'END' FOR;
EX:'END' A;

  A(30,30,1);
'END';
```

```
        PROGRAM PART(OUTPUT)
        INTEGER ZZ(100),NN(100),MM(100),II(100),P(100)
        INTEGER M,N,I,Z,S
        M = 30
        N = 30
        I = 1
        S = 1
  1     Z = 1
  2     P(I) = Z
        IF (Z - N) 10,20,10
  20    PRINT 60,(P(K),K=1,I)
  60    FORMAT(1X,40I3)
        GO TO 30
  10    ZZ(S) = Z
        NN(S) = N
        MM(S) = M
        II(S) = I
        S = S + 1
        M = Z
        N = N - Z
        I = I + 1
        GO TO 1
  30    S = S - 1
        IF (S) 31,31,32
  32    Z = ZZ(S) + 1
        N = NN(S)
        M = MM(S)
        I = II(S)
        IF (Z - M) 2,2,30
  31    STOP
        END
```

```
   VAR P: ARRAY [1..100] OF INTEGER;
   PROCEDURE A(M,N,I: INTEGER);
      VAR Z,K: INTEGER;
   BEGIN FOR Z := 1 TO M DO
         BEGIN P[I] := Z;
            IF Z = N THEN
            BEGIN FOR K := 1 TO I DO WRITE(P[K]);
                  WRITE(EOL); GOTO 1
            END ;
            A(Z,N-Z,I+1)
         END ;
 1:END ;
BEGIN A(30,30,1); WRITE(EOL)
END .
```

```
'BEGIN' 'COMMENT' COUNT CHARACTERS IN FILE;
   'INTEGER' I,J,Z42;
   'INTEGER' 'ARRAY' C[-1:63];
   'FOR' I := -1 'STEP' 1 'UNTIL' 63 'DO' C[I] := 0;
    EOF(60,L2); Z42 := 2^42;
L1: INCHARACTER(60,
 '('ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+-*/()$= .,'[]:#!%&^"<>@\_;')',
   I);
   C[I] := C[I] + 1;
   'GOTO' L1;
L2: 'FOR' I := 0 'STEP' 1 'UNTIL' 63 'DO'
    OUTPUT(61,'('A,9ZD/')', I*Z42, C[I]);
'END' :
```

```
'BEGIN' 'COMMENT' COUNT CHARACTERS IN FILE;
   'INTEGER' I,J,Z42;
   'INTEGER' 'ARRAY' C[-1:63], A[1:80];
   'FOR' I := -1 'STEP' 1 'UNTIL' 63 'DO' C[I] := 0;
    EOF(60,L2); Z42 := 2^42;
L1: INPUT(60,'('80(A)')',A);
      'FOR' J := 1 'STEP' 1 'UNTIL' 80 'DO'
      'BEGIN' I := A[J] // Z42; C[I] := C[I] + 1;
      'END' ;
   'GOTO' L1;
L2: 'FOR' I := 0 'STEP' 1 'UNTIL' 63 'DO'
    OUTPUT(61,'('A,9ZD/')', I*Z42, C[I]);
'END' ;
```

```
      FORTRAN IV PROGRAM COUNT(INPUT,OUTPUT,TAPE1=INPUT)
      INTEGER A(80), C(64)
      DO 10 I=1,64
   10 C(I) = 0
   20 READ 100,(A(J), J=1,80)
      IF (EOF,1) 50,30
   30 DO 40 J=1,80
      I = A(J)+1
   40 C(I) = C(I)+1
      GO TO 20
   50 DO 60 I=2,64
      K = I-1
   60 PRINT 201, K, C(I)
  100 FORMAT (80R1)
  201 FORMAT (1H ,R1,I10)
      STOP
      END
```

```
! COUNT CHARACTERS IN FILE "
   VAR I: CHAR;
      C: ARRAY [CHAR] OF INTEGER;
BEGIN FOR I := EOL TO ';' DO C[I] := 0;
      READ(I);
      REPEAT C[I] := C[I] + 1; READ(I)
      UNTIL EOF(INPUT);
      WRITE(' EOL',C[EOL], EOL);
      FOR I := 'A' TO ';' DO WRITE(' ', I, C[I], EOL)
END .
```

of the rich data-structuring facilities of PASCAL (cf. section *The description of objects during compilation*) which ensured a problem-oriented and machine-independent, but at the same time efficient and economic representation of data. Thus, PASCAL proved to be in many respects a necessary and ideal tool to formulate complex and large system programs.

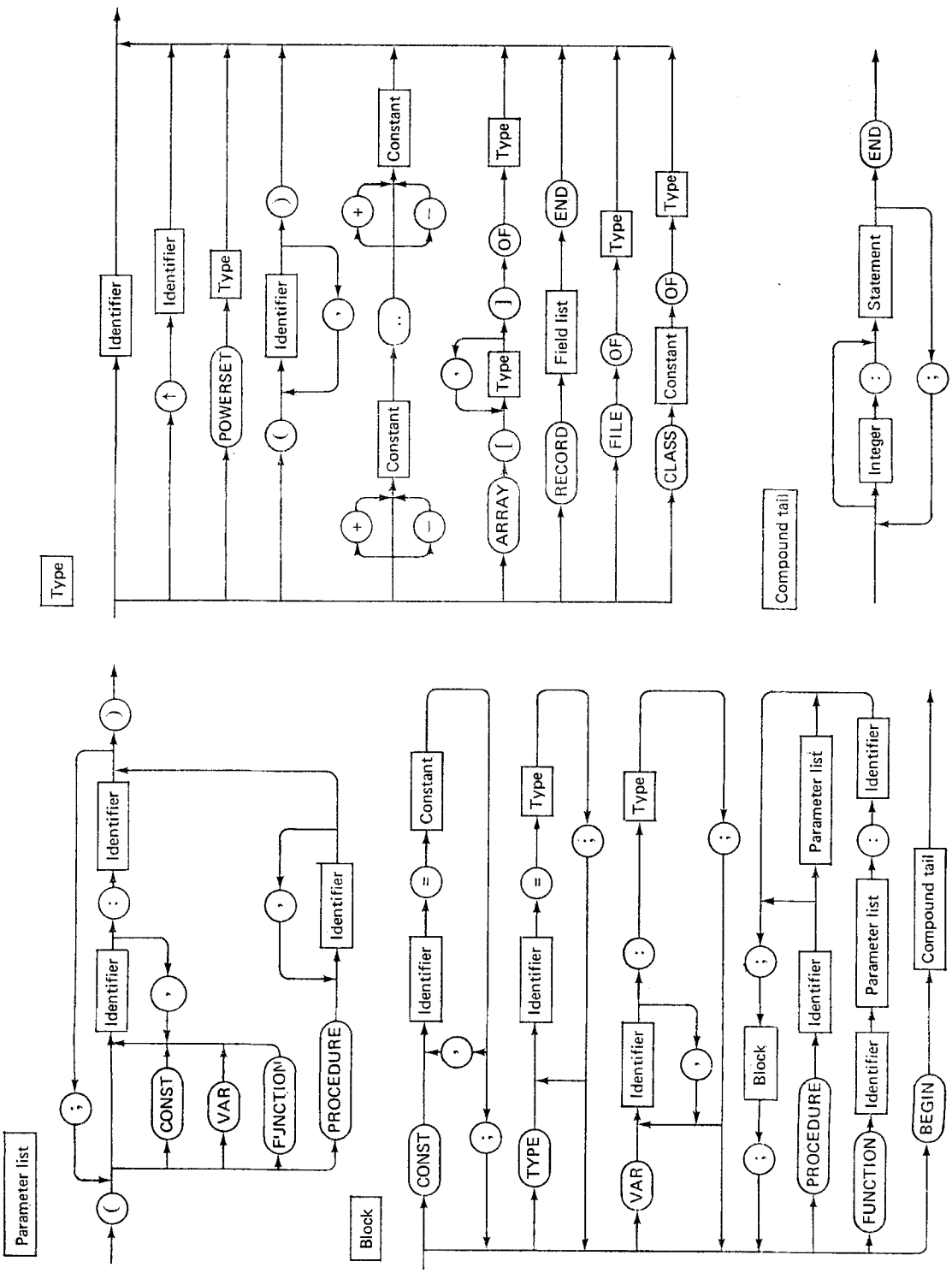
ACKNOWLEDGEMENTS

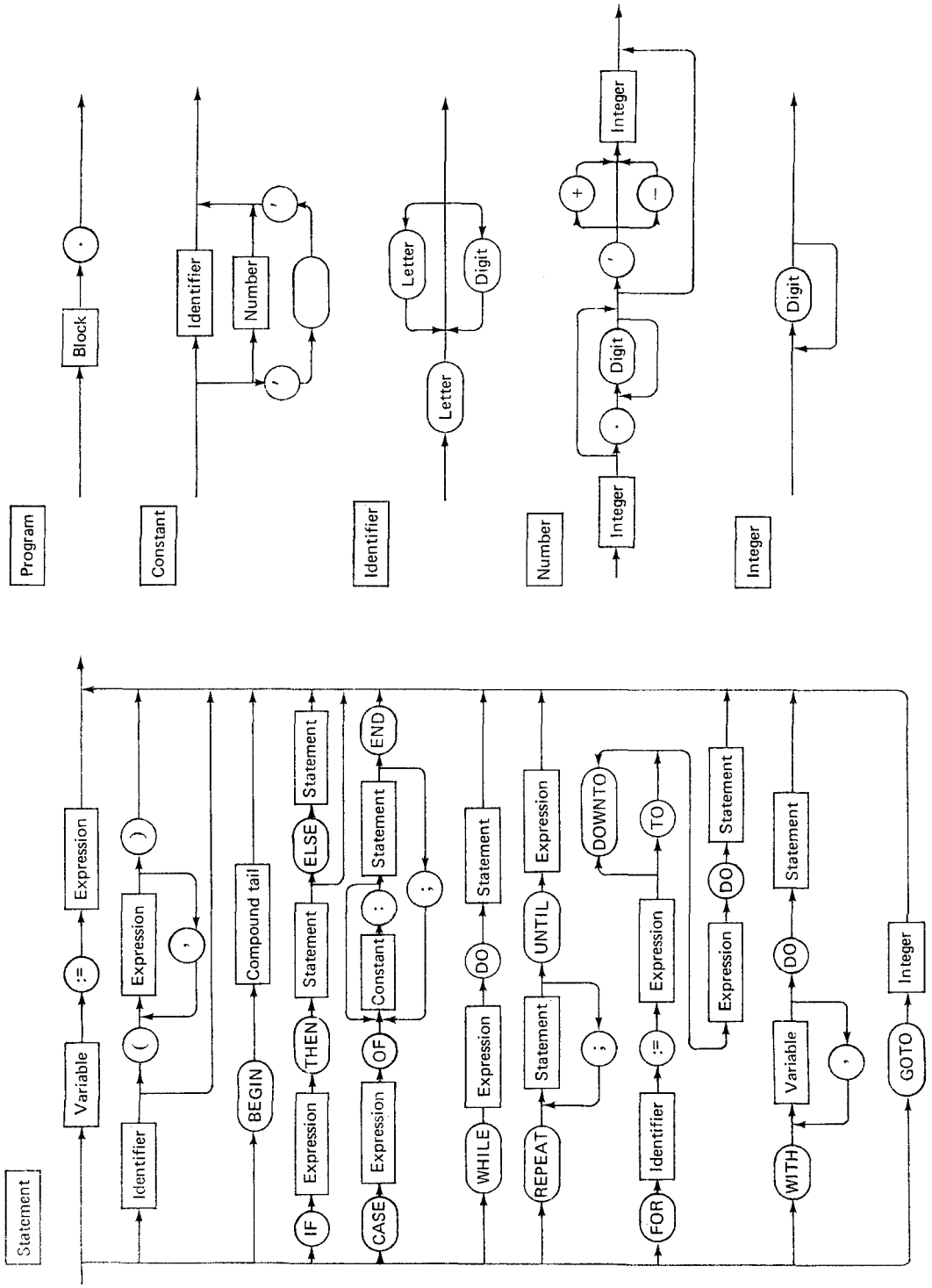
The author wishes to express his gratitude to his collaborators U. Ammann, E. Marmier and R. Schild, whose personal incentives and efforts were a necessary requirement for the successful development of the entire PASCAL project.

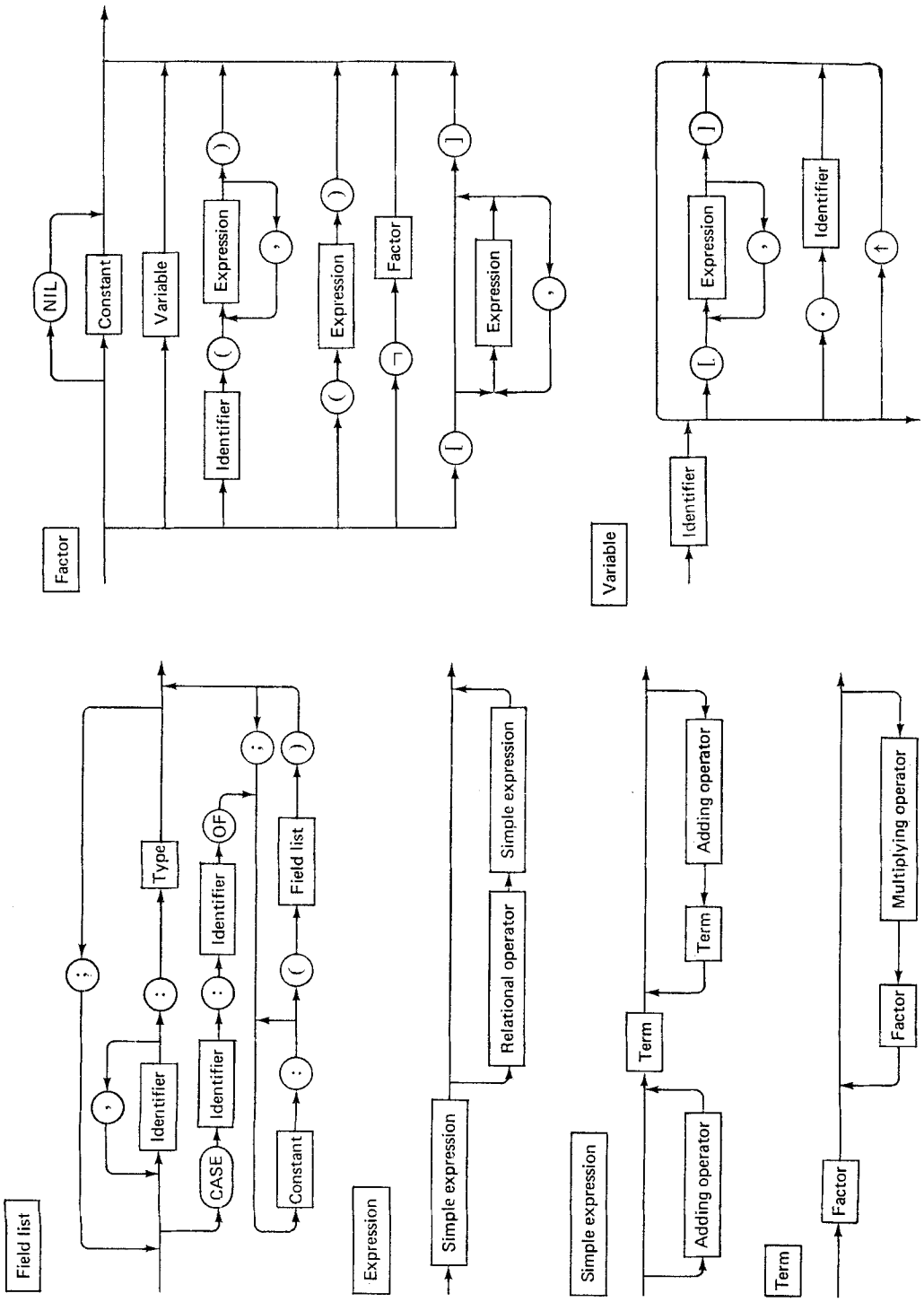
REFERENCES

1. N. Wirth, 'The programming language PASCAL', *Acta Informatica*, **1**, 35–63 (1971).
2. N. Wirth, 'Program development by stepwise refinement', *Comm. ACM*, **14**, No. 4, 221–227 (1971).
3. E. W. Dijkstra, *Notes on Structured Programming*, EWD249, Technical University Eindhoven, 1970.
4. E. W. Dijkstra, 'Ein ALGOL-60 Uebersetzer für die X1', *MTW*, **8**, 54–56, 115–119 (1961).
5. B. Randell and L. J. Russell, *'ALGOL 60 Implementation*, Academic Press, London, New York, 1964.
6. C. A. R. Hoare, *Set Manipulation*, ALGOL Bulletin, 27.3.4, 1968.
7. C. A. R. Hoare, *File Processing*, ALGOL Bulletin 25.3.3, 1967.
8. Control Data Corporation, *SCOPE Reference Manual*, Publication No. 60189400.
9. Burroughs Corporation, *Extended ALGOL Reference Manual*, Publication No. AA 37688.
10. M. E. Conway, 'Design of a separable transition-diagram compiler', *Comm. ACM*, **6**, 7, 396–408 (1963).
11. D. E. Knuth, *Top-down Syntax Analysis*, International Summer School on Computer Programming, Copenhagen, 1967.
12. D. E. Knuth, 'An empirical study of FORTRAN programs', *Software—Practice and Experience*, **1**, 105–133 (1971).
13. H. Weber, 'A microprogrammed implementation of EULER on IBM system/360 model 30'. *Comm. ACM*, **10**, 9, 549–558 (1967).
14. C. A. R. Hoare, 'The ELLIOTT ALGOL programming system', in *Systems Programming*, Ed. P. Wegner, Academic Press, London, New York, 1964.

APPENDIX: SYNTAX DIAGRAMS




Program
Block
.
Constant
Identifier
Number
'
Identifier
Letter
Digit
Number
Integer
+
-
Integer
Digit
Statement
Variable
:=
Expression
Identifier
(
Expression
,
)
Compound tail
BEGIN
IF
Expression
THEN
Statement
ELSE
Statement
CASE
Expression
OF
Constant
:
Statement
;
END
WHILE
Expression
DO
Statement
REPEAT
Statement
;
UNTIL
Expression
FOR
Identifier
:=
Expression
TO
Expression
DO
Statement
DOWNTO
WITH
Variable
,
DO
Statement
GOTO
Integer


Factor
NIL
Constant
Variable
Identifier
(
Expression
)
,
(
Expression
)
¬
Factor
[
Expression
,
]
Variable
Identifier
[
Expression
,
]
.
Identifier
↑
Field list
;
,
Identifier
:
Type
CASE
Identifier
:
Identifier
OF
;
Constant
:
(
Field list
)
Expression
Simple expression
Relational operator
Simple expression
Simple expression
Term
Adding operator
Term
Adding operator
Term
Factor
Multiplying operator
Factor