Pascal — The Language and its Implementation
Edited by D. W. Barron
© 1981, John Wiley & Sons, Ltd.

7

The Zurich Implementation

U. Ammann

1. Introduction

Pascal was developed by Niklaus Wirth at the Federal Instute of Technology in Zurich (ETHZ) in 1970 [1]. During the years of 1970 and 1971, parallel with the development and the definition of the language, the first Pascal Compiler–for the CDC 6000 computer family–was written at the Computer Science Institute of ETHZ [2]. This experience, on the one hand, led to the definition of a *revised* language [3] and, on the other hand, to the decision to write a new compiler from scratch. Thereby, both the language and the implementation matured from the previous project.

The bootstrap of the original compiler was done by means of SCALLOP–a medium-level programming lanaguage for CDC computers. First, the compiler was written in a subset of unrevised Pascal. Then it was hand-translated into SCALLOP and bootstrapped. Finally, the compiler was extended to accept the full unrevised Pascal.

The second compiler was started in the summer of 1972 and completed about two years later. It was developed with the aid of the original Pascal compiler using the commonly known bootstrapping technique. This subdivided the whole task into the following four phases:

(1) Programming the new compiler in unrevised Pascal.
(2) Compilation of the result of phase (1) with the old compiler.
(3) Translation by hand of the result of phase (1) into revised Pascal.
(4) Compilation of the result of phase (3) by means of the compiler obtained in phase (2).

This bootstrap can be depicted with T-diagrams as shown in Figure 1. The compilers resulting from phases (1) to (4) are therein marked with the respective numbers. Concerning this bootstrap, it is worthwhile to note the following:

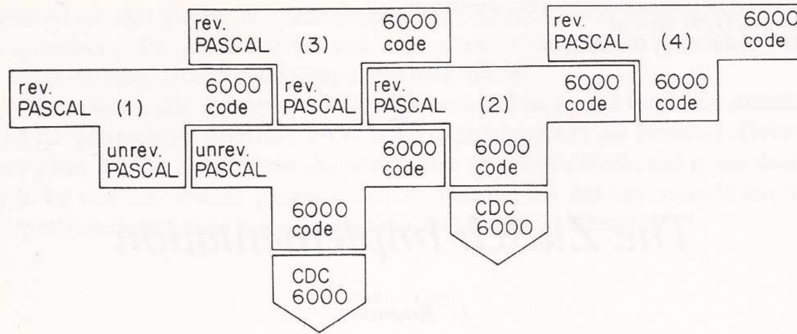

Figure 1. The bootstrap of the new Pascal compiler

(a) The old and the new compiler differ greatly from one another. This is not so much because of the language revisions but rather because the new compiler generates better code. To reach this goal a reorganization of the compiler in its code-generation parts was necessary and led to a more complex program structure.
(b) The compiler resulting from phase (2) obviously compiled revised Pascal. Nevertheless, it was in two respects not satisfying the demands. First, it had still been written in unrevised Pascal and hence could not profit from the language revisions. Second, it had been compiled by the old compiler, which did not translate into sufficiently efficient code. Therefore, the compiler speed of compiler (2) was rather moderate and its storage requirements were quite high.
(c) Thanks to the relatively few language changes the hand-translation in phase (3) of the bootstrap proved to be a more or less negligible task.
(d) With the completion of phase (4), the influence of the old compiler on the new one finally vanished.

A compiler bootstrap as performed and described above usually runs into several difficulties. Among them are:

(a) The implementation languages (in this case, revised and unrevised Pascal) have to be powerful enough to express programs of the complexity of a compiler with elegance and ease.
(b) The resulting compiler has to produce sufficiently good code. Otherwise it will become a victim of its own shortcomings. To be competitive, it must be neither slow nor spacious.

On the other hand, writing a compiler in a high-level programming language has many advantages: a saving in programming time, a low error rate, easy error detection, especially in the case of source-oriented error messages, and self-documen-

tation—not to speak of (hopefully) improved reliability, modularity, and maintainability.

Furthermore, when a compiler is written in the language it compiles, an additionaly advantage is that improvements in code generation often happen to pay back after a bootstrap. In spite of a larger source text, the compiler is quicker and uses less memory than before. This feedback is, of course, most welcome to every implementor.

During the development of the new compiler, the author came to know with persuasive power all of the above-mentioned advantages. In addition, it became clear that Pascal is not only relatively easy to compile but is also a powerful language which is perfectly suited for use as a compiler implementation language.

2. Syntax analysis and error recovery

Syntax analysis is done by applying the commonly known method of recursive descent. Each syntax diagram is associated with a (generally recursive) compiler procedure, which analyses the corresponding syntactic entity. Pascal supports this elegant method since it can be analysed backtrack-free with a minimal one-symbol look-ahead. However, some of the syntax diagrams [3] could not be taken as a model for the corresponding compiler procedures. On the one hand, this is because a distinction is made between several kinds of identifiers (constant, type, variable, field, procedure, and function identifiers) although they do not syntactically differ from one another. For the diagrams where these identifiers appear (i.e. in factor, simple type, and statement) the compiler procedures are based on a slightly different syntax. On the other hand, the somewhat overloaded diagrams, block and statement, have been divided up for separate implementation. Figure 2 shows part of the modified syntax diagram statement.

This top-down goal-oriented syntax analysis is completed by a bottom-up source-oriented symbol scanner (called *insymbol*) which, with every call, returns the next terminal symbol found in the input. Following the recommendations of Jensen and Wirth [3] word symbols are represented by *reserved words*. They are actually sorted by length and stored in a linear table. The words with equal length appear in order of decreasing occurrence. Surprisingly, this organization allows for a sufficiently efficient answer to the question of whether a letter sequence is a reserved word or an identifier. Identifiers may be of any length but must differ amongst the

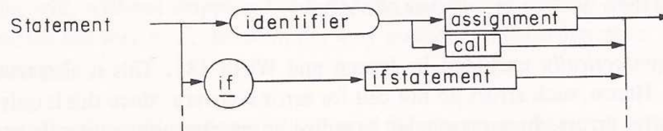

Figure 2. Modified syntax diagram statement

first ten characters, if their difference is to be recognized. The following structure shows the compiler program after hierarchically joining the compiler procedures:

- **program** *pascal*
  - **procedure** *insymbol*
  - **procedure** *block*
    - **procedure** *constant*
    - **procedure** *type*
      - **procedure** *simpletype*
      - **procedure** *fieldlist*
    - **procedure** *labeldeclaration*
    - **procedure** *typedeclaration*
    - **procedure** *variabledeclaration*
    - **procedure** *proceduredeclaration*
      - **procedure** *parameterlist*
    - **procedure** *body*
      - **procedure** *statement*
        - **procedure** *variable*
        - **procedure** *call*
        - **procedure** *expression*
          - **procedure** *simpleexpression*
            - **procedure** *term*
            - **procedure** *factor*
        - **procedure** *assignment*
        - **procedure** *compoundstatement*
        - **procedure** *gotostatement*
        - **procedure** *ifstatement*
        - **procedure** *casestatement*
        - **procedure** *whilestatement*
        - **procedure** *repeatstatement*
        - **procedure** *forstatement*
        - **procedure** *withstatement*

In order to keep the syntax as simple as possible, several obviously incorrect constructions as, for example,

if $i + j$ **then** S or **var** *ch*: '*A*'..4

are not *syntactically* excluded in Jensen and Wirth [3]. This is also true for the compiler. Hence, such errors do not call for error recovery, since this is only invoked by syntactic errors. In a reasonable organized compiler only syntactic errors may lead to a skip of part of the input—a rule which was strictly adopted by this compiler. If, for example, in a particular program a procedure is declared with fewer

parameters than are supplied at its call, then the surplus parameters are nevertheless analysed completely instead of simply ignoring the program text up to the closing bracket. Unfortunately, most compilers do not make this clear distinction between semantic and syntactic errors!

The treatment of syntactic errors can be done on two levels, namely, within the scanner on the level of characters and, outside it, on the level of symbols. In the implementation presented here error recovery is restricted to the second level, i.e. within the scanner no characters are deleted or inserted. This kind of error correction would only be useful in connection with specially marked word delimiters. The additional redundance of the distinguishing mark would then allow correction of trivial misspellings (e.g. 'BIGIN' could relatively easily be recognized as misspelled *begin*).

As to the error recovery, i.e. the error repair on the symbol level, a reliable, systematic, and economic solution was sought. Its description follows.

Let T denote the set of all terminal symbols of the language extended by the two symbols '*eoi*' (*end of input*) and '*other*' (characterizing a non-Pascal symbol), both of which are also returned by *insymbol*:

$$T = \{\textit{Pascal terminal symbols}\} \cup \{\textit{eoi, other}\}$$

Let S denote the set of compiler states, where each state is defined by the run-time stack (more precisely by the return addresses to be found there) together with the current value of the program counter. The treatment of (syntactic) errors now consists of extending in a sensible way the state transition function $st\ (S \times T \to S)$ to become a total function. Analogously, the function $pr(S \times T{*} \to T{*})$ which defines the program remainder left for analysis when passing from a state to its successor has to be made total.

The systematics of error recovery now results from associating each compiler state $s \in S$ with a set of terminal symbols R $(=r(s))$, the symbols which are relevant when the compiler is is state s. Should the symbol t returned by the symbol scanner happen to be unacceptable in the current state, then:

(a) An error message is emitted (whereby in the listing the position of the error is unambiguously marked with an arrow; compare Figure 3.

(b) The program remainder is skipped until a symbol $t' \in R$ is reached. The compiler and the function $r\ (S \to P(T))$ are tuned in such a way that the compiler will, without skipping t' advance to a state s' in which t' will be accepted. On the way to s', the compiler may possibly emit further error messages.

Hence, each phase of skipping is followed by the acceptance of at least one symbol and the compiler recovers immediately from any syntax error.

The quality of error treatment obviously stands and falls with the choice of the

function r and the state s' to be reached after the syntax error. Any choice of r will necessarily find itself between the two following extremes:

(a) $r(s) = A(s) \cup \{eoi\} \forall s$ where $A(s)$ is the set of symbols which can be accepted in s. Error repair in this case consists of ignoring the program remainder until an acceptable symbol (or *eoi*) follows ($s' = s$).
(b) $r(s) = T - \{other\} \forall s$. In this method no symbol will ever be ignored. However, reaching state s', in which t ($=t'$) will be accepted, is very problematic.

Clearly, neither of the two extremes is of any importance in practice. They would both lead to a catastrophic error recovery. In this implementation R is always composed of a set R' of global relevance–which is passed by parameter to the called procedure–and a set of locally relevant symbols L depending on the local error situation: $R = R' \cup L$. Hence, no globally relevant symbol is ever skipped. R is initially equal to the set *eoi*, **label**, **const**, **type**, **var**, **procedure**, **function**, **begin**, **goto**, **if**, **while**, **repeat**, **for**, **with**. Besides *eoi* which necessarily belongs to this set, it contains all the word symbols that unambiguously open a declaration or a statement.

In order to keep the compiler as simple as possible, error recover is always completely up to the called procedure (rather than to the calling one). For the called procedure this means:

(a) It cannot make any assumption regarding the program remainder which was left to it for analysis.
(b) It has to guarantee to the calling procedure to leave a program remainder which starts with a globally relevant symbol (i.e. a symbol R').

In this way, the task of the calling procedure is much simplified. The treatment of the errors does not take place at the calling site but rather completely within the called procedure. This simplifies the compiler considerably, since the number of calls significantly exceeds the number of procedures.

The compiler procedure *ifstatement*, giving an idea of the simplicity of this kind of error recovery, is shown below in the treatment of syntactic errors (pseudo-Pascal Notation). It is to be noted that, due to the one-symbol look-ahead, the first symbol of every syntactic entity is already read at entry to the associated compiler procedure. Analogously, the first symbol not belonging to the entity has been read at exit.

```
procedure ifstatement (R': set of T):
begin expression (R' + {then, else});
      if symbol = 'then' then insymbol
                          else error(20); (*then expected*)
      statement (R' + {else});
      if symbol = 'else' then
        begin insymbol; statement(R') end
end
```

```
000006 PROGRAM SYNTAXERRORS(OUTPUT)
000235    VAR I: INTEGER;
***       ↑14
000236    FOR I := 1 TO 10 DO
***         ↑18,17
000016      WRITELN(SIN(I);
***                        ↑4
000034 END.


                    ERROR SUMMARY:
                    **************


     4:  ´)´ EXPECTED
    14:  ´;´ EXPECTED
    17:  ´BEGIN´ EXPECTED
    18:  ERROR IN DECLARATION PART
```

Figure 3. Compilation of a syntactically incorrect program

In the compiler, every error message is coded by a number. The numbers of the errors encountered during compilation are retained in an error list (a packed Boolean array). At termination of the compilation of erroneous programs, the compiler calls an external routine with the error list as parameter. This routine overlays the compiler code with a procedure also written in Pascal. This procedure now writes out the error messages associated with the respective error numbers. This allows the output of arbitrarily detailed error messages without affecting the storage requirements of the compiler. Figure 3 shows the output resulting from the compilation of a syntactically incorrect program.

3. Semantic analysis and treatment of context-sensitive errors

The task of the compiler with respect to semantic analysis is threefold. First, it has to build up internal descriptions of the data structures defined in the user program. Second, the identifiers occurring in the program have to be retained together with their attributes. Third, the information obtained in this way has to be used to check the program for semantic errors.

As to the compiler-internal description of identifiers and data structures, the following is worth saying beforehand. Tables of *fixed* length have the decisive disadvantage that they are too big for most of the programs, but are too small for a few of them. In the former case they are uneconomical while in the latter case they even forbid compilation! Therefore entries for identifiers and data structures are *dynamically* created when needed (through a call of new).

At the end of the compilation of each block, the entries which have been made during compilation of the block are no longer needed. They are therefore eliminated from the heap by a call of the (non-standard) procedure *release*. Of course, this

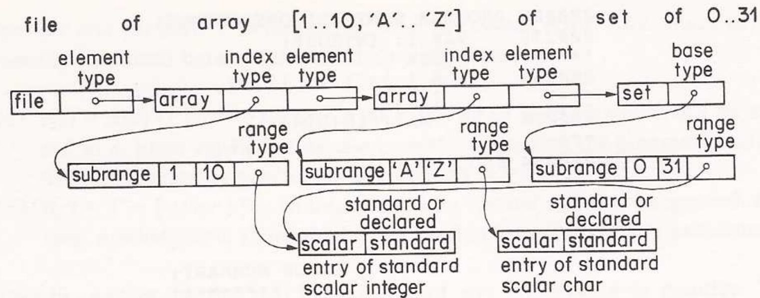

Figure 4. Example of compiler-internal description of data structures

could also be done by repeatedly calling the standard procedure *dispose*, but since the compiler uses the runtime heap as a second, explicit stack, the former solution is advisable.

To retain the attributes of standard and user defined identifiers and data structures within the compiler, a record is the appropriate data structure. It allows a suitable representation of the information collected during declaration analysis. The entries describing the standard identifier and data structures are made in an initialization phase preceding the declaration proper.

4. The internal representation of Pascal data structures

According to the three simple types and the four structuring possibilities, seven kinds of entries are distinguished: **scalar**, **subrange**, **pointer**, **set**, **array**, **record**, and **file** entries. The data structure chosen for their representation is therefore a *variant* record uniting seven variants. Combinations of the structuring concepts are quite naturally represented by linked entries. In this overview the exact definition of the variant record cannot be given. For details, the interested reader is referred to Amman [4, 5]. Here a hopefully self-explanatory example may suffice (Figure 4). Each entry is therein represented by a box, the partitions of which each stand for a field of the record variant. The appearance of the tag field as the first field of the variants should not confuse the reader. In fact, there exist fields common to all variants (as well as further fields specific to the variants); they are left out for the sake of simplicity. For example, common to all variants is a field defining the storage requirements associated with the data structure in question. This field will be discussed later.

5. The representation of identifiers and their associated attributes

As mentioned earlier, six kinds of identifiers can be distinguished. These are the **constant**, **type**, **variable**, **field**, **procedure**, and **function** identifiers. Accordingly, the

record for internal description of identifiers and their associated attributes contains six variants.

The character code (of the first ten characters) of a name and the reference to its type entry are two obvious fields of this record and are common to all variants. The type reference is a pointer to the description of the data structure associated with the identifier (for function identifiers it points to the result type; for procedure identifiers the pointer is nil).

To satisfy the demands for a fully dynamic name table, all name entries on the same declaration level are united to form an alphabetically ordered binary tree (in post-order). In this data structure, making entries and searching for a particular entry are both quite efficient operations. As experience shows, it is not even necessary to balance the trees (during or after completion of the analysis of a declaration part) since degenerated trees are rare–especially in large programs where they really might affect compile speed. Eliminating a whole tree is done in the trivial way described earlier.

Hence, two further fields of each name entry contain references to the left and the right subtree respectively. The roots of these binary trees are stored in a compiletime display. During declaration analysis, the attributes associated with each name are first united in an appropriate record variant. Then this record variant is–under preservation of the alphabetical order–entered as a leaf into the tree of the local entries. When the attributes of a name are requested, the display is used to access the roots of the trees in the right order. First, the tree containing the local names is searched; there then follow the trees containing the global names and finally the one in which the descriptions of the standard names are stored.

During compilation of a **with** statement the root of the binary tree in which the entries of the fields of the selected record are united is loaded on top of the display. (This root is a variant-specific field of the structure entry describing the record.) Thus it is guaranteed that during the compilation of a **with** statement names are always first searched in the record opened last.

Figure 5 shows by example how names and their attributes are retained in the compiler. As in Figure 4, the attribute records are symbolized by boxes. Again only the attributes which are relevant in this context are displayed. For details the reader once more referred to Ammann [4, 5].

In one-pass compilation, names usually have to be completely defined before the first usage. Two cases of non-removable forward references in which it is impossible to observe this rule are given below:

```
type a = record pb: ↑b;          procedure p;
         end;                       begin q end;
     b = record pa; ↑a;          procedure q;
         end                        begin p end;
                                 begin p; q end
```

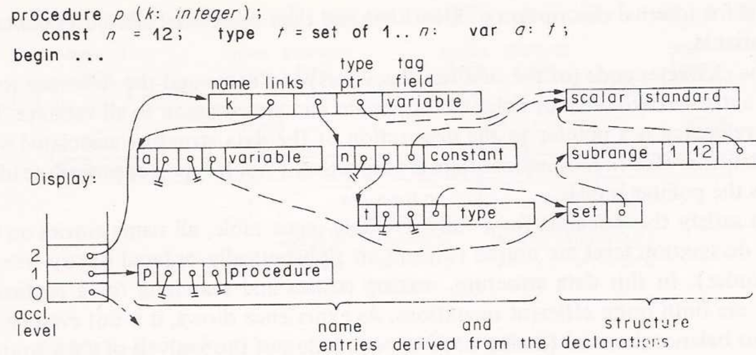

Figure 5. Example of the representation of names and their attributes

Therefore, the following two exceptions are provided:

(a) Pointer type declarations are allowed to reference type names which are defined later–but have to appear in the same block and on the same declaration level.

(b) Procedures (and functions) may be declared forward. In such cases, the block of the procedure is replaced by the letter sequence 'forward'. Later, the declaration proper has to follow–again in the same block and on the same declaration level as the forward declaration. The declaration proper must neither repeat the parameter list nor a possible function result type. These forward declarations follow the expositions in the *User Manual* but are not included in the *Report* [3].

6. The treatment of context-sensitive errors

Pascal is a type-oriented language. Its strong type concept–an established means to improve security in programming–gives the compiler the opportunity (and the duty) to recognize a variety of context-sensitive errors. It was tried to achieve the following three main goals:

(a) Recognition of as many errors as possible at compile time and concise indication of these errors to the user.

(b) Sensible reaction of the compiler to the use of undeclared identifiers and multiple declarations of the same name.

(c) User-friendly interpretation of the rules of type compatibility within the limits imposed by the report.

To illustrate these three points a program containing a few semantic errors is given in Figure 6. A bad compiler would probably not report all of these errors or

```
000006 PROGRAM SEMANTICERRORS(OUTPUT);
000235    CONST N = 10;
000235    TYPE P = ↑T1;  T1 = N..9;  T2 = 1..N;
***                             ↑102
000235    VAR R1: RECORD F1: T2; F2: CHAR END;
000237        R2: RECORD G1: 1..15; G2: CHAR END;
000241        A: ARRAY [0..9] OF INTEGER;
000253    PROCEDURE P(M, N: T2);
***                 ↑160,101
000005       VAR PP: P;
000006    BEGIN END;
000020 BEGIN R1 := R2;  R1.F1 := 12;
***                              ↑303
000021       P(1,2*A[N] - 1,A[R2.G1] + K[N].X + 1 DIV 0)
***          ↑103     ↑302 ↑126        ↑104           ↑300
000037 END.


                    ERROR SUMMARY:
                    **************

  101:  IDENTIFIER DECLARED TWICE
  102:  LOW BOUND EXCEEDS HIGHBOUND
  103:  IDENTIFIER IS NOT OF APPROPRIATE CLASS
  104:  IDENTIFIER NOT DECLARED
  126:  NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION
  160:  PREVIOUS DECLARATION WAS NOT FORWARD
  300:  DIVISION BY ZERO
  302:  INDEX EXPRESSION OUT OF BOUNDS
  303:  VALUE TO BE ASSIGNED IS OUT OF BOUNDS
```

Figure 6. Example of the treatment of context-sensitive errors

would postpone their recognition until runtime (102,300,302,303). The undeclared name K gives rise to only one error message (104), since the operations of indexing and field selection could well be legal if an appropriate declaration of K was supplied. This is why they are—*in dubio pro reo*—not diagnosed. Also, in both places in which the doubly defined identifier P is used, the right declaration is eventually supplied. As a consequence, the parameter list is completely analysed. Particularly, in the third parameter which is accurately recognized as being superfluous (126), two additional errors are found (104,300). As an example for the user friendliness in the interpretation of type compatibility the assignment between two records is given ($R1 := R2$). It is tolerated since the assignments between corresponding components ($R1.F1 := R2.G1; R1.F2 := R2.G2$) are legal. Again, the exact marking of errors is to be noted which allows for rapid location and correction of each error.

7. Address generation

In the preceding sections those aspects of compilation have been treated which are independent of the target machine. What has been said so far therefore permits

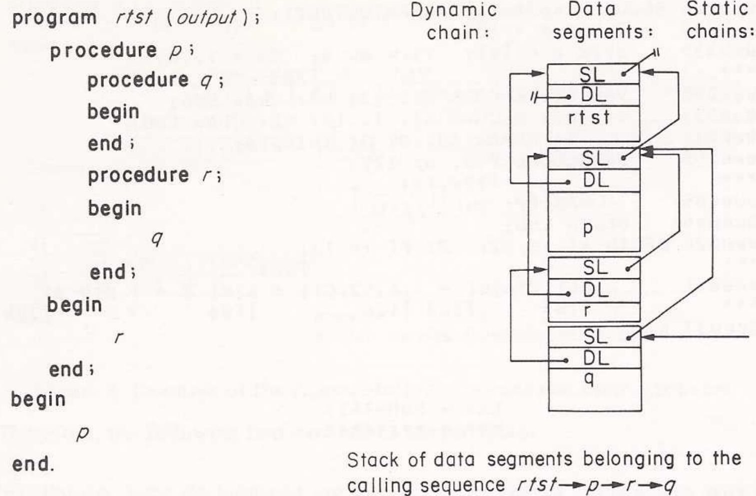

Figure 7. Dynamic and static link

unrestricted comparison with other implementations of Pascal. This is now no longer the case since all decisions concerning the design of address and code generation are dominantly influenced by the architecture of the target machine.

The local data of every procedure (and function) are united in a data segment. Addressing is relative to the segment origin. For run time a stack containing the data segments of all activated procedures is provided. Pushing and popping of data segments (at procedure entry and exit) is achieved by means of the dynamic link (DL) which chains every data segment to its immediate predecessor in the stack. Variable addressing is done through the static link (SL). It chains only those data segments which–according to the scope rules of Pascal–are currently accessible. SL and DL are incorporated in the head of every data segment (in the first and second words respectively; see Figure 7).

As Figure 8 depicts, every data segment is in general dividend into six logically distinguishable areas: segment head (containing SL and DL), function result, parameter descriptors, parameter copies, local variables, and 'anonymous' values–which appear in this order of succession.

The main store memory of the CDC 6000 series (as well as the 7600, Cyber 70, and 170 models) has 60-bit words and no provision for partial word addressing. Hence, efficient packing of data structures which are declared *packed* is a necessity, particularly with respect to the fact that the compiler itself profits much from such a storage economy (mainly with name and structure entries).

A field common to all seven variants of a structure entry is a record defining the storage requirements associated with the structure described by the entry.

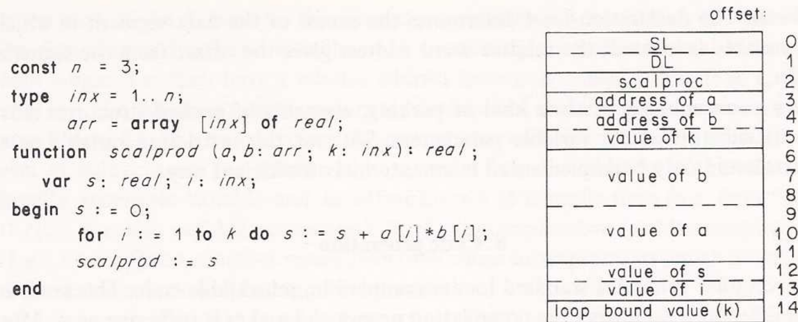

Figure 8. Function declaration and data segment derived therefrom

```
size: packed record words: addressrange; (*0..131071*)
                    bits: bitrange (*0..59*)
      end
```

Within packed arrays and records as many elements as possible are packed into one word. However, elements using less than one word of memory never cross word boundaries. This restriction simplifies address calculation without influencing the packing density too much.

Of course, access to elements of packed data structures is somewhat slower and needs more code. However, it allows a considerably more compact representation of data. For example, the compiler requires 16.5K words of code and when it compiles itself (in sixty-four central processor seconds) it needs up to 4.5K words of heap (besides maximally 2K words of stack). With unpacked entries the compiler code is 16K words in length, and to compile itself the compiler needs 65 CP seconds and up to 9.5K words of heap! Figure 9 shows an example of the mapping of a packed data structure into the memory.

Hence, record field addresses in general consist of a word address and a bit address (e.g. ⟨1,17⟩ for *f*4). Both of them are stored as attributes of the field name. The address attribute of a *variable* name is a pair ⟨*declaration level, relative word*

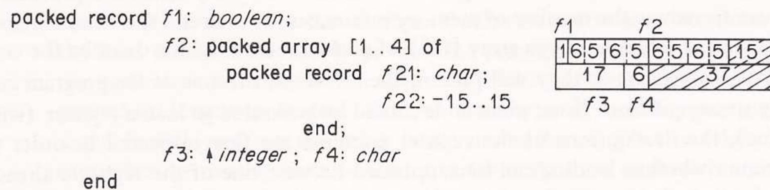

Figure 9. Memory assignment of packed data structures

address⟩. The declaration level determines the access of the data segment in which the variable is located; the relative word address gives the offset from the segment origin.

As a consequence of theis kind of packing, elements of packed structures must not be substituted for variable parameters. Without this restriction variable parameters could only be implemented in an extremely inefficient way.

8. Code generation

The compiler generates standard loader compatible, relocatable code. This saves an assembly phase following the compilation proper and makes it truly one-pass. After the compilation of every procedure (function) the code is written onto the code file. At that moment, the code is nearly complete, the only exceptions being references to other procedures and jumps leading out of the procedure. However, both of these kinds of external references are satisfied at load time by the loader/linker. All other fixups to the code, in particular forward jumps within the procedure, are done by the compiler. But since the code file (as any other Pascal file) is strictly sequential, the code of every procedure must be retained in core until completion of the compilation of its body. Again, in order to allow for a fully dynamic solution, code segments (for 150 words of code each) are allocated in the heap when needed and are chained to a linear list. The code stored in this list is written out at the end of the procedure declaration and the memory so far used by the list elements is returned to the Pascal system.

9. Descriptions of the register contents

The CDC 6000 hardware offers three classes of eight registers each: index (so-called B) registers, address (A) registers, and operand (X) registers. A and B registers consist of 18 bits; X registers are 60 bits each. A and X registers are coupled in a rather strange way, in order to load the contents of a memory location into register Xi $(1 \leqslant i \leqslant 5)$. Ai is set to its address. To store the contents of a register Xj $(6 \leqslant j \leqslant 7)$ in a memory location, Aj has to be set to its address. Solely $A0$ and $X0$ are independent of each other. Since there are no other instructions addressing the memory, even the simplest assignment needs three instructions: loading into Xi, copying from Xi to Xj, and storing of Xj. This fact obviously calls for a preventive measure to reduce the number of memory references. To this end the compiler contains three variables (of type **array** [0.7] **of packed record**) which describe the contents of all registers as they will present themselves at runtime of the program currently in compilation. Now, when code should be generated to load a register (with a value), the descriptions of the register contents are first inspected in order to determine whether loading can be suppressed becuase one of the registers already contains the value in question.

The compiler only remembers those register contents which are easily described.

In the case of the X registers these are the constants and two classes of 'variables'. The first class includes the variables that are directly accessible from the data segment origin, i.e. they have a relative address known at compile time (e.g. i, $a[2]$, $r.x.f$, but also function results, parameter descriptors, and anonymous values; (compare Figure 8). The second class consists of the variables addressed through one level of indirectness. The relative address results from the addition of the value of a directly accessible variable and an offset known at compile time (e.g. $input\uparrow$, $p\uparrow$, $p\uparrow.f[2]$, but also variable parameters). In all other expressions (as for example, $a[i]$, $p\uparrow.g\uparrow$, $i + r.x.f$) the compiler remembers only those subexpressions which are of one of the above-mentioned distinguished forms.

It is noteworthy that in packed structures the number of items remembered is multiplied by the number of fields packed into one word. When for example, the first word of a variable of the type given in Figure 9 is loaded, $f1$ and $f2$ are in register. Every succeeding reference to any of the nine components can then be established without memory reference. Register bookkeeping pays back much more in the presence of data packing!

Part of the description of every non-free X register is a reference counter which contains the current number of references to the register. The references originate from the descriptions of the subexpressions currently in compilation; e.g. right before the compilation of the miltiplication in $a[i] * i$, the description of the register containing the value of i has a reference counter of 2. When no references exist, it is interesting to know for how long the register has not been referenced. The measure for this is the value of the compiler-interval instruction counter at the moment of the erasure of the last reference. The longer the register has not been referenced, the less interesting it is to the compiler to preserve its contents from being overwritten. When an X register is needed for code generation, the compiler scans through their descriptions trying to find a free one. If none exists, a register, the contents of which are known to the compiler, has to be released. The decision as to what register's contents should be destroyed depends, on the one hand, on the time elapsed since the erasure of the last reference and, on the other, on the kind of contents. Constants are regarded to be of less importance than the variables which generally cannot be reloaded with the same ease.

Two things are worth noting in connection with the effort to reduce memory references by the aid of register content descriptions:

(a) The intention to reduce the number of store instructions, as was done with the load instructions, was given up. The principal difficulty with delaying the generation of store instructions lies in the fact that the memory is in general no longer up to date. This, on the one hand, forbids a post mortem dump facility and, on the other hand, forces the compiler to take special care when generating code to load a variable. As soon as there is a chance that the variable to be loaded coincides with a register which has yet to be stored, the delayed store operation must take place before the loading.

(b) The parameter descriptors of the first four parameters are always passed in *X* registers (and not in memory). This has not only the advantage of providing for more compact code at the calling site but also allows the compilation of every procedure (or function) to start with non-empty register contents.

In the case of the *A* registers, the compiler remembers exactly those addresses which belong to the variables that are remembered.

The *B* registers are mainly used as base addresses for the data segments, namely *B*2 for the main program variables and *B*5 for the local variables of a procedure (or function). Hence, access to these data segments is always direct (using *B*2 and *B*5 respectively). Whenever access to a data segment of intermediate level first occurs, it is achieved by following the static link. However, if there is a *B* register which is currently available, this register is immediately used to point to the origin of the data segment in question. Subsequent accesses to this data segment are then direct too, i.e. without the detour along the static link. There is a good chance that indirect addressing of data segments is thereby considerably reduced.

The old Pascal compiler, which had no register content descriptions, obviously generated much superfluous code. When compiling the new compiler, it generated approximately 23K words of code. This number was reduced to about 16.5K words when the new compiler was used to compile itself.

10. The compilation of control statements

It is characteristic of all control statements that they have a distribution point D (from which the computation flows out in several directions) and a concentration point C (in which the computation flows together from several directions). Distributors and concentrators are of great importance to the bookkeeping of the register contents. When the compiler reaches a distribution point it must locally store the currently valid register content description in order to be able to reload it when needed. This is necessary because the compilation of these statements is strictly sequential while the execution is not. Nevertheless, the compilation of any branch starting in D should profit from the register contents which are valid when the distributor is reached. For matters of simplicity the descriptions of all register contents are cleared when the compiler reaches the concentrator. Figure 10 depicts the maintenance of the register content descriptions during the compilation of an **if** statement.

The most elementary control statement is, of course, the **goto** statement. Here is another argument to support those who consider it harmful. In the terminology introduced in this section, labels represent concentrator points. These concentrators are, however, far more awkward than the concentrators in regular control statements. The reason for this is that, at the labels, computation can flow together from arbitrary directions. Hence, it is impossible, without a complete analysis of

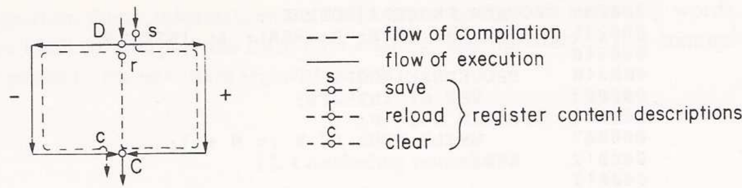

Figure 10. Compilation of an **if** statement

the computation flow, to find a non-trivial register content description which is valid when a label is reached. Therefore, all contextual knowledge has to be discarded at a label, making jumps to these labels inherently expensive operations.

11. The post-mortem-dump facility

When a runtime error is detected in a user program, a post-mortem dump (PMD) procedure is optionally called. This procedure not only informs the user about the exact place in his program, where the error was detected, but also about the kind of the error (e.g. division by zero, cost limit exceeded). In addition, all activated procedures of the user program are traced. First, the name of the procedure is written out together with an indication of the source line on which it was called. Then the names of all unstructured local variables and parameters appear with the values they had when the error was detected.

Figure 11 gives an example. It shows a program containing an infinite loop. When this program is executed and runs into cost limit, the CDC operating system will emit an appropriate dayfile message. Control then returns to the Pascal runtime system which calls PMD.

To allow for a correct interpretation of the runtime stack, local variables and parameters of every procedure *p* are each described by two-word entries. The first word contains the (display code of the) name of the variable; the second word includes its address relative to the segment origin and information on its type. These two-word items are optionally appended to the code of *p*. A zero entry acts as a terminator. To make this data available to PMD, the code of *p* is preceded by another two-word entry, the first of which contains the name of the procedure while the second points to the entries described above (see Figure 12).

When PMD is activated it follows the dynamic chain and interprets the data segments one after the other. A field in the segment head allows PMD to decide whether the associated procedure *p* was called formally. If so, its entry address is found in the head of its data segment (it was stored there when *p* was called). Otherwise the return address is extracted from the segment head. It points to a (code) word which is naturally preceded by the word containing the instruction calling *p*. From this instruction the entry address can be obtained.

After determination of the entry address the name of *p* and the pointer to its

```
000006 PROGRAM PMDTEST(OUTPUT);
000235    VAR CH: CHAR; R: REAL; I: INTEGER;
000240
000240    PROCEDURE LOOP;
000003       VAR N: INTEGER;
000004    BEGIN N := 0;
000007       WHILE TRUE DO N := N + 1;
000012    END;
000017
000017 BEGIN CH := 'A'; R := 1E-20; I := 20;
000020    LOOP;
000021 END.

  - PROGRAM TERMINATED AT: 000007 IN LOOP

    LOOP           AT: 000020 PMDTEST

    N               =    305101

    PMDTEST

    I               =    20
    R               =  1 0000000000000E-20
    CH              =    A
```

Figure 11. Post-mortem dump

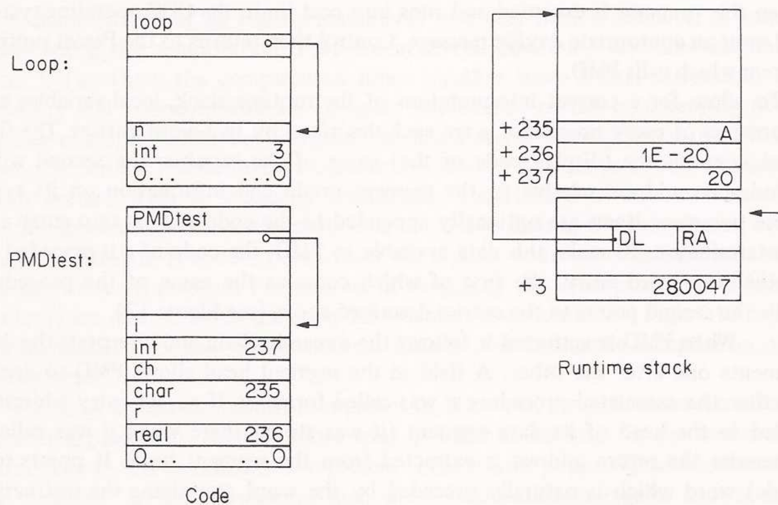

Figure 12. Interpretation of data segments by PMD

post-mortem dump information are easily found in the two preceding words. This allows PMD to interpret the local data segment appropriately. After commpletion PMD passes to the next data segment (if any).

12. Concluding remarks

The overview given in this paper is incomplete. For more details the interested reader is referred to Ammann [4, 5, 6].

Currently the compiler described here is running on about 150 installations all over the world. Its user community is steadily growing. Except for the kernel of the runtime support (0.5K words including data) the whole system is written in Pascal. This comprises

(a) the compiler (nearly 7400 source lines),
(b) the I/O procedures performing the conversion between character strings and the internal representation of numbers,
(c) the procedure which writes out the compilation error messages, and
(d) the post-mortem dump procedure.

The fact that all these system components could be written in Pascal within about fourteen man-months shows that this language is very well suited for systems implementation. However, the experience gained in this project with one-pass compilation was somewhat contradictory. On the one hand, it proved to have its merits, above all ease of implementation amd modest I/O activity. On the other hand, it imposes severe restrictions on the quality of the generated code and suffers from relatively high storage requirements. In this respect a multipass compiler represents a better solution: the passes can easily be overlayed and an optimization pass provides for a really efficient code. Unfortunately, Pascal does not support overlay structures! Comments from users of the implementation are generally positive. Amongst the most frequently cited advantages are modest compilation time, sensible error messages, comparatively efficient code, (optional) post-mortem dump, and dense, user-controllable packing of data. Occasional complaints mostly concern the absence of certain runtime tests (e.g. the tag-field test), the restriction of sets to single machine words, the suppression of structured variables (notably records) in the post-mortem dump, and the unsafe connection to externally compiler routines. Future releases will hopefully improve the system in these respects.

Acknowledgements

The author is indebted to Ann Duenki, Professor Niklaus Wirth, and Dr. Rudolf Schild for carefully reading through the manuscript and suggesting many improvements.

References

1. Wirth, N., The programming language Pascal, *Acta Informatica*, **3**, 35–63, 1971.
2. Wirth, N., The design of a Pascal compiler, *Software–Practice and Experience*, **1**, no. 4, 309–334, 1971.
3. Jensen, K., and Wirth, N., *Pascal–User Manual and Report*, 2nd ed., Lecture Notes in Computer Science, No. 18, Springer-Verlag, Berlin, Heidelberg, New York, 1975.
4. Ammann, U., *The Method of Structured Programming Applied to the Development of a Compiler* (Eds. A. Guenther *et al.*), International Computing Symposium 1973, pp. 93–99, North-Holland, 1974.
5. Ammann, U., *Die Entwicklung eines Pascal-Compilers nach de Methode des Strukturierten Programmierens*, ETHZ Dissertation Nr. 5456, 1975.
6. Ammann, U., On code generation in a Pascal compiler, *Software–Practice and Experience*, **7**, no. 3, 1977.