SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 929–942 (1981)

The Implementation of Case Statements in Pascal

ARTHUR SALE

Department of Information Science, University of Tasmania, Box 252C GPO Hobart, Tasmania, Australia 7001

SUMMARY

This paper examines the probable implementations of the case statement in Pascal, and analyses their consequences. A particular implementation for the Burroughs B6700/7700 series illustrates the necessary investigations. The techniques, although not new, are brought together to serve as a stimulus to improving the implementation of case-statements in other compilers for Pascal, and to provide ideas for implementors. The architectural limitations of even high-level computers in implementing the case statement are briefly discussed.

KEY WORDS Case statements Computer architecture

PURPOSE

This paper seeks to document the implementation techniques which can be used to implement the Pascal case statement, and the decisions that were made in an implementation for the Burroughs B6700/7700 series.

THE PROBLEM

The implementation of a case statement breaks into two distinct parts. One is the set of *actions* (statements) between which the case-selection is to choose: these may be simply compiled according to their nature and followed by an unconditional branch which leads to the statement following the case statement.

The other major part is the *selection*: the evaluation of the case expression and the identification of which action is to be initiated as a consequence, followed by a transfer of control to that action. The paper is solely concerned with implementing this part of the case statement, and deals with issues of efficiency, range of efficacy, and correctness of implementation.

THE PASCAL-P HERITAGE

Many compilers for Pascal are based on the compiler for the CDC Cyber and 6000 range (Pascal-6000), or on a semi-portable version producing code for an imaginary stack machine (Pascal-P). As a consequence, the treatment of case statements in these compilers has often survived unchanged into their descendants, even though it is not optimal. The technique used is simple, and can be briefly summarized.

(a) Both compilers, and their descendants, make a single pass through the source text, generating code as they go.

0038–0644/81/090929–14$01.40

Received 30 May 1979
Revised 19 February 1981

(b) On meeting the **case** keyword, code is compiled to evaluate the case expression, followed by an unconditional branch over the actions to part (d).
(c) The actions are compiled as they are encountered, each followed by an unconditional branch to the next executable statement after the case statement. Each case-label constant is stored in a linked list together with the corresponding code address of the action, and the list is maintained in ascending order of the constants.
(d) On meeting the closing **end** keyword, the selection is compiled. Since all case-label values are now known, the expression is checked to ensure that its value lies within the limits of the least and greatest constant values, followed by an indexed branch through a jump-table to the appropriate action.

The jump-table entries which do not correspond to any case-label value in the source text, and a case expression value outside the case-label limits, usually cause an error-termination. Some compilers implement an otherwise clause as an extension, when such events cause control to be passed to it. A few compilers treat such events as empty actions and control passes to the next statement.

This implementation technique has several machine-independent consequences:

(a) The time taken for the selection is practically independent of the width of the selection and the values of the case-label constants. There may be arithmetic and other second-order variations, but for all practical purposes the relationship is:

$$\text{time for selection} = k_1$$

where k_1 is a constant.
(b) the code-space required for the selection is made up of a fixed part (the checking and indexing) and a part whose size is proportional to the difference between the greatest and least case-label values. Let this difference be called r, and let k_2 and k_3 be further constants:

$$\textit{code space} = k_2 + (k_3 \times r)$$

(c) Due to this code-space proportionality, the implementor may put a limit on the size of the table. In the event of this limit being exceeded, the case statement is disallowed. If the implementor does not take this step, then such statements either cause other errors ('PROCEDURE TOO LONG') or some reasonable statements consume an unreasonable amount of resources. An example to evoke these effects is the following, adapted from the Pascal validation suite of programs.[1]

```
case i of
    1000 : j := j+1;
   -1000 : j := j-1;
end;
```

THE EFFICACY OF AN IMPLEMENTATION

To judge an implementation of a language feature, we must first ask if it exactly models the abstract notion embodied in the language feature. The implementation outlined earlier meets this criterion, otherwise it would have been unacceptable. The next question we should ask is over what range of validity the modelling holds, and whether any implementation errors could arise if this range is exceeded. Clearly the technique described is limited by its space demands, and many Pascal compilers refuse to compile the example given earlier. However, provided the range-check on the case-expression is never omitted, and provided that there are no modulus arithmetic effects in indexing into the jump-table, no erroneous compilations can occur.

The initial implementation in the University of Tasmania B6700 compiler[2] used this technique, but gave an error message if the range exceeded 300, implying a jump-table of 150 words. This limit was somewhat arbitrarily chosen so as to encompass the char type (256 values) as the type of the case expression, and to avoid a power of two. The limit was subsequently raised to another arbitrary value of 1000, and there is no good reason why it could not approach the limit of 2^{14} set by the addressing range of the unconditional branch.

One might query whether, given such a range, the technique is in any way limiting. After all, the counter-example given is highly artificial. However, experience with this implementation over several years brought a number of complaints derived from just this source: programmers wanted to do something they considered reasonable, but were prevented by the implementation. Examples are given in an Appendix.

Having satisfied oneself as to the correctness of the implementation and its range of efficacy, only then should the third question be asked: How efficient is it? The characteristic feature of the jump-table technique is that it is generally the fastest available for selection between many values. Any alternative whose selection time grows with the width of the selection must eventually become slower.

However, there are implementation techniques which are faster for very few alternatives, and there are implementations which use less space for some case statements. The almost unquerying acceptance of this technique is therefore somewhat puzzling, especially as older languages (for example BCPL[3]) have had more sophisticated compilers which employ a variety of stratagems.

We are therefore left with the conclusion that the jump-table technique is unduly limiting, and is not always the most efficient in time of code-space.

THE COMPARISON TECHNIQUES

An immediately obvious alternative is to compile the selection as a linear sequence of comparisons:

```
if e = c1 then goto label1;
if e = c2 then goto label2;
if e = c3 then goto label3;
.....
```

The time to accomplish a selection now depends on the value of the expression, and is a minimum for the first value and a maximum for the last. If a comparison takes time α and a **goto** time β, and all case-label values are equiprobable, then the average selection time is:

$$\textit{time for selection} = \frac{(\alpha+\beta)+(\alpha n+\beta)}{2}$$
$$= \beta + \frac{\alpha}{2}n$$

Since the overhead (β) is low, this technique may be faster than the jump-table technique for selection between very few alternatives.

The code-space required is simply proportional to the number of case-label values n (not their range):

$$\textit{code space} = k_4 \times n$$

Obviously, whether this is better or worse than the previous technique depends on the sparseness of values in the case-label range.

A more sophisticated technique is the if-tree: the selection is accomplished by a nested set of comparisons organized into a tree. If the selection problem is considered to be a selection of a subrange of values, including the subranges representing values which do not occur as case-labels (rather than as a selection between individual values), then this can easily be coded by a recursive procedure which splits its total subrange in two and calls itself to handle the two parts. Some optimization is possible when only a few alternatives remain, or by omitting comparisons which can be known to succeed at the leaf level. In the face of such optimizations detailed analysis is difficult. It is probably sufficient to note that:

$$\textit{time for selection} \simeq k_5 \times \log_2 m$$

$$\textit{code space} \simeq k_6 \times m$$

where m is the number of subranges, and is bounded by:

$$1 \leqslant m \leqslant 2n+1$$

These two techniques were used, for example, in a BCPL code-generator for implementing the **switchon** statement on an IBM 7090 series. An interesting twist is the use of tree-nodes of degree 3 which rely on the CAS instruction. The mnemonic CAS is not an example of IBM prescience: it stands for *Compare Accumulator with Storage* and produces conditional skips depending on whether the comparison yields less, equal, or greater.

The comparison techniques are inherently correct and have no implicit limits apart from excessive time or space usage for very complex case-statements. Special range-checks are not needed, unless the comparisons are somehow limited by modulus arithmetic or other effects.

It should be noted that the problem of selecting an action within a case-statement can be transformed into a decision table. Consequently all the techniques for compiling 'optimal' code for decision tables may be employed for compiling case statements.[4] However, case statements are simpler in structure than a general decision table, and the approach given above achieves satisfactory efficiency with a relatively simple algorithm.

TABULAR TECHNIQUES

If memory is a scarce resource, one should consider how the memory requirement for selection can be pared to a minimum. About the least that one could get away with if the case-label values cannot be implicit (as in a jump-table) is a table which holds records of the following form:

```
var
    searchtable = packed array[1. .n]of
                    packed record
                      caselabel : selectortype;
                      action    : addressoffset;
                    end;
```

It is then necessary to code a search algorithm to search this table for a particular case-label value, and to branch to the indicated address. The algorithm may be a linear search through the table, a binary search through a table ordered at compile-time, a hash lookup in a suitably created table, or any other alternative you can think of. The code may be in-line, or it may be shared amongst case statements by incorporating it as a system procedure.

For example, a 1972 BCPL compiler for the CDC 1700 mini-computer was able to choose between the jump-table technique, the linear table search with a tightly coded loop, and a binary table search. The binary table search was implemented as a procedure call to a system procedure which 'returned' to the indicated address if a match was found.

Other table organizations are possible; for example a heap-organized one (cf. Reference 5) as in the Heapsort algorithm. What differentiates all these tabular methods from the jump-table technique—which is naturally a tabular method too—is the explicit inclusion of case-label values in the table, and the dependence of the table size and selection time on some measure loosely connected to the complexity of the case statement, rather than the range of the greatest and least case-label values. Users of compilers are much more ready to accept the rejection of a highly complex construct than a simple one.

THE TASMANIA B6700 COMPILER—A CASE STUDY

The jump-table technique

In reviewing the case statement implementation for the B6700/7700 compiler maintained by the University of Tasmania, two principles were used to guide the choices:

(a) No case statement of reasonable complexity should be rejected by the compiler.
(b) The resource demands of any reasonable case statement should not be excessive.

The meaning of 'reasonable' and 'excessive' is left vague; nevertheless it was felt that the jump-table technique fails one or the other of these tests for sparse selections over a wide range of case labels.

However, before considering alternatives in detail, the jump-table technique was analysed. The jump-table itself consists of half-word (3 bytes) unconditional branches, and must be half-word synchronized. The range-check and indexed branch add up to 23 bytes of instructions on the assumption that the range limit values are fitted into 8-bit literals, and there is a 0–2 byte padding required to achieve the jump-table synchronization. The range-check is never omitted as the consequences of a wild branch which lands outside the jump-table are potentially disastrous. If the range is r, the space requirements are therefore:

$$\begin{aligned} \textit{code space} &= \textit{overhead} + \textit{padding} + \textit{jump-table} \\ &\simeq 24{\cdot}5 + 3r \text{ bytes} \end{aligned}$$

The execution time is much more difficult to estimate, particularly in the B6700/7700 series where three factors complicate matters: the asynchronous operation and differing memory speeds, the different model characteristics, and difficulty of access to timing data.[6, 7] However, to illustrate the comparison, consider the execution to be dominated by memory accesses and measure it in units of this size. Instruction fetches will require four word-accesses for the overhead, plus another to access the table. Data accesses are owing to the necessity to evaluate and store the expression value. The execution time is derived on the assumption that all case-label values are equiprobable; the worst-case is approximately double this estimate. Case-label values are assumed to fit in 8-bits. Violation of this assumption only affects the formulae given, not the correctness or range of validity of the code.

```
<evaluate expression>
NAMC(local temporary)              {address of un-named cell}
STON                               {store, but leave on stack too}
<load case-label value>            {a constant}
EQUL                               {compare}
BRTR(address)                      {if true, then branch}
{ ---------------------------------end of first selection}
VALC(local temporary)              {retrieve copy of expression}
<load case-label value>            {a constant}
EQUL                               {compare}
BRTR(address)                      {if true, then branch}
{ ---------------------------------repeat above section as needed}
BRUN(default handler)              {unconditional branch}
```

code space $= \textit{first case} + \textit{other cases} + \textit{default}$

$= 9 + 8(n-1) + 3$

$= 4 + 8n$bytes

time for selection $= \textit{instruction fetches} + \textit{data fetches}$

$$= \frac{8n+5}{12} + \frac{n}{2}$$

$= 0{\cdot}83 + 1{\cdot}83\,n$ memory cycles

A comparison tree

The first step in deriving a comparison tree is to consider the list of case-label values as creating an ordered list of subranges, each with its unique target address. For example, in the following statement:

```
case i of
  1,2,3 : s1;
  7,8 : s2;
  10 : s3
end;
```

the subranges are as follows:

subranges	target action
$-maxint$ . . 0	*default*
1 . . 3	*s*1
4 . . 6	*default*
7 . . 8	*s*2
9 . . 9	*default*
10 . . 10	*s*3
11 . . *maxint*	*default*

Each tree node then corresponds to a break between intervals; if there are m intervals then there are $m-1$ breaks. If the tree is constructed straightforwardly but so as to minimize the code-space required, then the optimum tree for the above case-statement is shown in Figure 1.

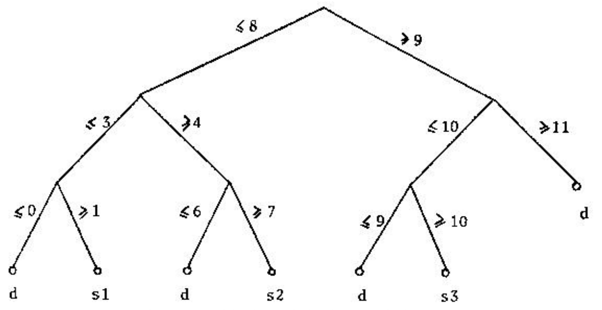

Figure 1. An example of a search tree

Most of the same considerations with respect to retrieval of the expression value apply to the generated code here as for the linear comparison method, and the same solution will be assumed. The code for each comparison is similar, with the substitution of some other test (GRTR for example) for the equality test. The required code-space is defined by the recurrence relations:

$$\begin{aligned} &\textit{code space}_2 &&= 11 \text{ bytes} && (m = 2) \\ &\textit{code space}_{m-1} &&= \textit{code space}_m + 8 && (\text{if } m \text{ even}) \\ &\textit{code space}_{m-1} &&= \textit{code space}_m + 11 && (\text{if } m \text{ odd}) \end{aligned}$$

An approximation is clearly

$$\textit{code space} \quad \simeq 9{\cdot}5(m-1) \text{ bytes}$$

The execution time can be similarly derived; a suitable approximation is:

$$\begin{aligned} \textit{time for selection} &= \textit{instruction fetches} + \textit{data moves} \\ &= (\tfrac{8}{6}+1) \times \textit{depth of search} \\ &\simeq 2{\cdot}33 \log_2 m \text{ memory accesses} \end{aligned}$$

Note that although the code space requirements still grow linearly with m, this method capitalizes on any subrange structure in the case-label values, and it is possible that the number of subranges is much less than the number of case-label values, or the range of values. Intuitively, this method seems to have resource demands which correspond fairly well with a programmer's notion of complexity of a case-statement. Since the time for selection grows slowly with the number of intervals, the method performs well for sparse selections of reasonable size.

Using the masksearch instruction

The B6700 architecture contains an instruction (SRCH) which is designed for searching tables. When provided with a descriptor pointing to a data segment, a one-word bit-mask, and a one-word comparison value, it linearly searches down the segment for an equality test under the mask pattern. The search starts from the top of the table (segment) and decrements. The result left on the top of the stack is the index of a successful match, or -1 if no match is found. Clearly, this can be used in an implementation of case-statements using a tabular technique.

One possibility of using this instruction is to set up a table where each word is a packed record containing an action address and a case-label value. Since the action address would have to be loaded to the stack and used as an indirect address by the dynamic branch instruction (DBUN), it is constrained to be a half-word address. Consequently a 14-bit address field would suffice to give the same addressing range as the byte-addressing branch instructions, leaving 34 bits for the case-label value in the word. The space requirements for each case-label value amount to one word (6 bytes) in the table, and from 0 to 2 bytes padding to half-word align each piece of the action code, making a total of 7·5 bytes/value on average.

An alternative is to devote a whole word to a case-label value. The index found is then used to construct an address in a jump-table within the code-segment. The two tables together require $(6+3) = 9$ bytes per case-label value; slightly higher than the previous method. It is then possible to dispense with range-checking completely, and let the SRCH instruction carry out the validity check. In the packed record scheme, range-checking is necessary to prevent expression values from masquerading as other values by hiding under the masking. Obviously the checking could *never* be omitted without creating the possibility of an incorrect compilation. The second method could therefore be expected to be slightly faster.

Our decision was that the second method was the more desirable, since it imposed no limits on the case-label values which would be admissible, and would quite happily compile even:

```
case i of
  maxint : writeln('  VERY POSITIVE');
  -maxint : writeln ('  VERY NEGATIVE')
end;
```

The difference in space requirements is not large (9 bytes per value against 7·5) and offset for small tables by the first method's greater overheads to carry out a range-check.

The characteristics of this implementation, and the code skeleton, are shown below.

```
  <evaluate expression>
  ZERO
  LNOT                            {creating an all-1s mask}
  NAMC(table descriptor)
  LOAD                            {load a copy of the table descriptor}
  SRCH                            {execute the search}
  <load address of table Lt>
  ADD                             {form halfword address}
  DBUN                            {dynamic (indirect) branch}
  <pad out to halfword boundary>
  BRUN(default)                   {branch if SRCH returns -1}
Lt: BRUN(....)
  BRUN(....)
  ....
```

code space $= \textit{instructions} + \textit{padding} + \textit{code table} + \textit{data table}$
$= 12 + 1{\cdot}5 + 3(n+1) + 6n$
$= 16{\cdot}5 + 6n$ bytes

time for selection $= \textit{overhead} + \textit{data accesses for average case}$
$= 4 + 0{\cdot}5n$ memory accesses (B7700)
$= 6 + 0{\cdot}5n$ memory accesses (B6700)

Performance comparisons

With methods with such varying characteristics, direct comparisons are difficult. With the exception of the two linear search techniques, no two methods have the same parameters. We can, however, measure the characteristics of a particular case-statement presented to a compiler, and derive from these some indicators of the best method to use for that particular statement. Clearly, for large sparse selections the comparison-tree technique will be appropriate, and for moderate-sized compact selections the jump-table will be good. The questions that the compiler-writer is interested in are:

(a) Whereabouts do the changeovers take place?
(b) Is code-space or execution-time to be the criterion?
(c) Are any methods of such low utility as to not be worth the trouble of writing the code-generation routines?

To explore some of these issues, consider a case-statement which has n distinct actions, each labelled by a case-label. The case-labels are adjacent (form a compact range), but are a subrange of a larger type. For small values of n, this will enable us to plot the resource requirements of each method, and this is the area of prime doubt where we do not know whether the linear search methods are of any benefit. The jump-table method is seen at its best (as there are no gaps in the table), and the search-tree method is seen as something approaching its worst (as all subranges are of length 1).

The results are shown as Figure 2, and are plotted parametrically on a space-time diagram.

From these results it is clear that the range of utility of the linear search methods is small. Under the given assumptions, the linear comparison method (ii) requires less space for $n \leqslant 4$, and executes faster for $n \leqslant 2$ on the B7700. Two-way case statements are rare, though of some use; one-way case statements have a very limited use.

Given the good execution speed performance of the jump-table method, dominating all others for large n, it is clear that it should be preferred as long as the space requirements are not excessive. The rapid increase in execution time with n for the linear methods indicates that they will not be of much use for large selections, and the only merit in implementing them must lie in selections between a few values sufficiently widely separated to make the jump-table unattractive. The frequency of such cases is probably fairly low.

The questions asked before can now be answered in reverse order:

(c) The only two methods incorporated into the University of Tasmania compiler are the jump-table and the search tree. The linear search methods do not have a sufficiently high range of utility.
(b) Since execution times of both methods are quite good, space considerations dominate.

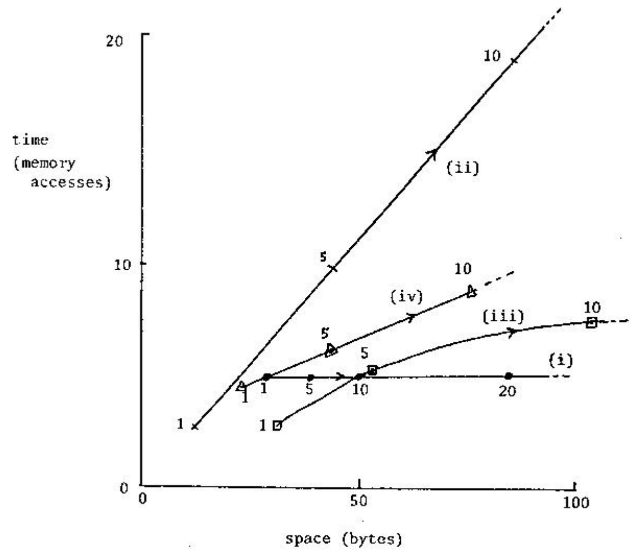

Figure 2. Performance of methods on a space-time map for a B7700. (i) jump-table, (ii) linear search, (iii) search tree, (iv) masksearch instruction.

(a) Since the jump-table technique is faster, use of the search tree will not be worthwhile until its space requirements are less than those of the jump-table. How much less to warrant the change is arguable. In the University of Tasmania compiler the changeover occurs when the space demands are equal so as to minimize the memory requirements.

CONCLUSIONS

Summary

In conclusion, it can be seen that a slight increase in complexity of a Pascal compiler leads to a much improved treatment of case statements, in that every case statement that would be considered to be of reasonable complexity by a rational programmer can be compiled. The efficient jump-table technique is retained for use where it does not require an excessively large table, whereas a search-tree may be used to capitalize on common destinations for adjacent selector values and to be reasonable for sparse selections.

It is surprising that so few Pascal compilers seem to have implemented anything other than the jump-table technique. A large measure of the responsibility for this state of affairs must be given to the Pascal-P compiler and its siblings, which have been widely copied. When a software product is good enough to warrant copying on the scale that these have evoked, then the shortcomings of the software are extremely likely to become frozen into its descendents. A very high degree of sophistication is thus demanded of high quality software. To counter this problem, a tested patch to the

Pascal-P compiler should be available to make it generate suitable P-code for a search-tree. Since the P-machine's instruction set is very low-level, and close to that of most architectures, this should encourage maintainers to improve the quality of their compilers.

Comments on machine architecture

Another consequence of this investigation is the clarity with which the complete unsuitability of current machine architecture for dealing with the case statement is revealed. Even on the Burroughs B6700/7700 architecture, supposedly the highest-level architecture available, the construct demands quite difficult decisions and complex code, and a specialized search instruction proves to be virtually useless for this purpose.

The case-statement, as a language feature, was invented comparatively recently.[8] Since the architecture of most extant machines is derived from tradition and convention ante-dating this invention, this lack is not particularly surprising. However, case statements are important in modern programming languages as a special kind of selection, and it is reasonable to ask whether special provision could be made. Four general directions suggest themselves. Note that in this discussion it is not proposed that the case statement be a directly executed instruction, but that facilities be provided to simplify its compilation and to facilitate correct and efficient execution.

In a similar manner, the **goto** survives as the branch at the machine architecture level because it is an efficient mechanism for implementing while statements, if statements, etc. (cf. Reference 9).

(a) Associative memories often seem to be cast in the role of Maxwell's demon: invoked to carry out any magical operation. However, associating a value (target address) with a key (selector value) is exactly what we want. In the absence of a direct parallel association, an association must be derived by algorithmic means, leading to the next possibility.

(b) An instruction similar to the B6700's SRCH could be implemented to retrieve an index from a data segment by some more sophisticated algorithm than a linear search. One possibility is to conduct a binary search on a table which is assumed to be ordered. Binary field operations on an index are very simple to implement in hardware. Another is to implement a hardware hash lookup, leaving the software to construct a suitable table with the same algorithm at compile-time.

(c) The jump-table technique, with an implicit key-value, is valuable and should be preserved. Its only drawback is the clumsiness of the range-checking and offset; it would be trivial to incorporate these into a descriptor for the jump-table, or design a checking instruction. It is vital to correctness of software that the check never be omitted, and be of low cost. The VAX-11/780[10] has an instruction of this type, appropriately given the mnemonic of 'CASE'. This machine is, of course, of very recent design.

(d) The search-tree could readily be automated by an instruction which searched a table which was implicitly tree-structured (as in the familiar Treesort algorithm). A one-bit tag in each table word could indicate either a comparison value or a target address, or the tag could be implicit.

Unavailable Statistics

A search of the literature yielded little factual data on how the case statement is used in practice. Several questions are of interest to implementors of languages containing such statements, amongst which the following deserve attention:

(a) How frequently are case statements used when there are no psychological or habitual barriers to their use?
(b) What is the distribution of the number of actions making up a case-statement?
(c) Since case statements are directly analogous to record variants in Pascal, how frequently is this construct used to select a variant in preference to a series of if-statements? (This will affect the proportion of very narrow and even one-way case statements.)
(d) How frequently do sparse case statements occur as to be inefficiently implemented by a jump-table? (Existing practice is no good guide if only jump-table techniques are available to programmers.)

A careful examination of Pascal programming style could be of considerable use to implementors, but is not currently available. The major difficulties seem to lie in capturing a representative sample of Pascal programs, and in avoiding influences of implementations or other programming languages.

APPENDIX

This appendix describes three of the plausible uses of the case-statement construct which have been brought to the author's notice. In two of the cases cited the programmers concerned remarked that they had adopted the particular construct because they were unaware of any implementation limits on its use, and that they were unlikely to do so again after they had discovered what happened. The conclusion can be drawn that 'unusual' uses of the case-statement will remain 'unusual' so long as the common implementations prohibit them; their rarity is not necessarily a consequence of a poor structure or abstraction.

The first case was caused by a program which was dealing with Roman numerals. In the midst of this program appeared a case-statement which had as case-constants the numeric values of the symbols in the Roman numeral system: 1, 5, 10, 50, 100, 500, 1000. Of course, there were other ways to carry out the selection required and indeed the subproblem it appeared in, but the use was plausible and consistent.

The second example occurred in a program with integer variables, and on a compiler which had an extension which permitted case-label values to be given as a range. The program fragment is self-explanatory:

```
case i of
  -maxint . . -1 : HandleNegativeValue;
  0:               HandleZeroValue;
  +1 . . +maxint : HandlePositiveValue;
end;
```

In the third example, the programmer had packed two characters into a single integer value, and wished to use the case-statement to identify command sequences. The following program fragment is not standard Pascal, but will explain the intent better than the original.

```
var twochars = packed array[1. .2] of char;
  . . . .
case twochars of
    'HE' : ProcessHelloCommand;
    'FL' : ProcessFlashCommand;
    'TU' : ProcessTurnCommand;
    'BY' : ProcessByeCommand;
end;
```

With the substitution of a string type for the Pascal packed array the above example is in fact allowed in a new language called S-Algol[11] and running under UNIX on PDP-11 machines.

REFERENCES

1. B. J. Wichmann and A. J. H. Sale, *Draft Validation Suite for Pascal Compilers*, preliminary draft 1978.
2. A. J. H. Sale, *B6700/7700 Pascal Reference Manual*, Department of Information Science Report R77-3, University of Tasmania, 1977.
3. M. Richards, *The BCPL Reference Manual*, Technical Memo No. 69/1-2, The Computer Laboratory, Cambridge, 1971.
4. U. W. Pooch, 'Translation of decision tables', *Computing Surveys*, **6**, 2 (1974).
5. D. E. Knuth, *The Art of Computer Programming—Vol 3 Searching and Sorting*, Addison-Wesley, 1973.
6. Burroughs Corporation, *Burroughs B6700 Reference Manual*, Form 1058633, 1969.
7. Burroughs Corporation, *Burroughs B7700 Reference Manual*, Form 1060233, 1975.
8. C. A. R. Hoare, *Hints on Programming Language Design*, Technical Report CS403, Computer Science Department, Stanford University, 1974.
9. D. Gries, *Compiler Construction for Digital Computers*, Wiley, New York, 1971, pp. 279-281.
10. Digital Equipment Corporation, *VAX-11/780 Architecture Handbook*, 1977.
11. A. J. Cole and R. Morrison, *An Introduction to S-Algol Programming*, University of St. Andrews, 1979.