SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 915–919 (1979)

Miniscules and Majuscules

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

THE PAST

Once, not so long ago, practically all programming was carried out in one case of letters: the upper case of majuscules A, . . ., Z. Necessarily, programs were printed this way, and readers of programs had to contend with it. Indeed programmers probably became some of the most fluent readers of capitals in the world, and they forgot that this was not normal.

Then along came the ROM, dot matrix printers, and the VDU, and it became possible to produce lower case, or miniscule, letters economically. Those of us who had been arguing that writing and reading in upper case alone was dismally bad at last found our arguments acceptable. Though perhaps it is too soon to say that the battle for more readable programs is over, it is reasonable to claim that victory is in sight. For example, a very pragmatic Computing Centre Director told me a few months ago that he refused to consider any terminal or printer devices unless they had both letter cases available.

But the freedom to use either case brings with it some important stylistic decisions. This note attempts to bring these decisions together, and suggest some preferred solutions. Some have been incorporated into the draft Pascal Standard.[1]

MIXING THE CASES

If writing programs entirely in upper case letters is bad, then writing programs entirely in lower case letters is foolish. The result is more readable than previous practice, but it throws away many possible improvements in style. However, once we allow programmers to mix letter cases in programs we find that we need to formulate a principle:

Principle 1
The meaning of a program should not be affected by the case of any letter in it, with the exception of literal constants containing character values.

This principle can be justified on three equally cogent grounds:

(a) The case of a letter does not conventionally change the meaning of a natural-language word. For example, this sub-clause contains the word *the* in four different representations: only the italic one has substantially different meaning. For me, this is THE most important justification.

(b) Much as we regret it, some installations are still restricted to one letter case, perhaps by six-bit characters. Consequently, for programs to be portable it must be possible to convert all letters to one case without causing the program meaning to change. Equally obviously, this cannot apply to character values themselves.

0038–0644/79/1109–0915$01.00
Received 1 March 1979

(c) If programming languages permit programmers to denote different objects by similar names, say x and X, then the chances of accidental error are increased and programs may be more difficult to read. We have all probably experienced the practical jokester who substitutes zeros for ohs, as in *ST0PFLAG* for *STOPFLAG*. Lower case may foil him, but we do not want to add new possibilities for confusion.

In fact, this principle has been expressed in the draft standard for the programming language Pascal, save for one tiny loophole which must be corrected in the next draft. The University of Tasmania Pascal compiler has always processed input text in accordance with the principle, by translating identifiers and reserved words into a canonic form for internal processing. Other techniques are possible which will still fit in with the principle: for example, rules restricting the use of cases to particular token classes, or prohibiting any representation of a word other than the one declared. To me, these seem somewhat *ad hoc* measures, and I prefer the generality of equivalence.

KEYWORDS AND IDENTIFIERS

I have written previously about keywords and identifiers[2] so I shall be brief here. When keywords are basic symbols distinct from the set of identifiers, as in Algol 60's reference representation, then distinguishing by the use of case is defensible, if regrettable. When, as in Pascal, keywords are simply identifiers with a reserved meaning, there is no good reason for treating them differently from user-defined identifiers.

The arguments language designers use in choosing between reserved words and basic symbols do not admit of any definite resolution in favour of either. It seems to depend on the number of keywords needed: large sets indicate a basic symbol approach may be preferable to avoid confusion. Experience with several old Algols, with examples of $BEGIN, 'BEGIN' and $\underline{b}\,\underline{e}\,\underline{g}\,\underline{i}\,\underline{n}$, indicates that care is needed. Subtle hints are sufficient for the machine. Perhaps distinguishing on the first letter of the word is the least objectionable. Personally I prefer to avoid the mess and use reserved words: I do not much care for user-names which conflict in appearance with the fixed-meaning words.

COMMENTARY

Comments are easy to dispose of. They are solely intended for the human reader, and comments therefore ought to be written to conform to all the usual conventions of prose text. Sentences ought to start with a capital and end with a full-stop, etc. Example in Pascal:

```
{At this point, the variable Index points to the record in the list that has the same
key as the search key. It remains to be determined whether this is the sentinel or a
correct lookup.}
```

This is a consequence of my next principle:

> *Principle 2*
> Upper and lower cases of letters should be used only in ways that make programs more readable and more understandable to people reading them.

IDENTIFIER NAMING

We come to the second major problem with the naming of identifiers. There seem to be four main classes of identifier names:

(a) mathematical symbols (x, i)
(b) single words (*count*, *table*)

(c) noun phrases (*x move, apple type*)
(d) verb phrases (*sort the vector, print day of the week*)
There are no serious problems with the first two classes: mathematicians and prose writers have always preferred to write symbols and words as I gave them in the examples: in pure lower case. (To be fair, since mathematicians have a paucity of symbols, capitals are sometimes used to denote special objects: for example B for a matrix, but b for an integer variable. With a wider range of identifiers, programmers need not resort to this.)

My classes (c) and (d) do give problems, for it is now widely recognized that space ought to have limited significance in programming, acting as a separator of tokens. Before I turn to the practices that have grown up in programming, let me look at natural languages. There are two main mechanisms provided for constructing aggregates from words:
in English, for example: hyphenation
apple-pie, king-hit, out-of-doors
in Nederlands and Deutsch, for example: concatenation
buitenshuis (out-of-doors, outside the house),
Taschenwörterbücher (pocket dictionary)
I am assuming that we are not interested in the agglutination method of stringing words together with intervening spaces since this makes lexical parsing much more difficult.

Someone once remarked (I have lost the reference) that programming languages sometimes look like a Continental revenge for English. There is some evidence for this view: for example, Algol 60 and Pascal make provision only for the concatenating form of agglutination, while COBOL and PL/I (originating from the other side of the Atlantic) make provision for the hyphenated agglutination. COBOL uses the hyphen and PL/I uses an underlined space.

If I assume that Nederlanders are quite happy to write long agglutinated identifiers (an assumption I cannot prove), it is certainly not true that English-speaking programmers are happy with it. Only short compounds are acceptable in English, and even these are often truncated. Identifiers such as *finddistinctionnode*, *modeerror* and *pixelarray* are not pleasant to read, let alone long procedure names such as *ensureexpressionvalueisonstack*. What remedies do we have?

In Standard Pascal, the only remedy is to utilize the two letter cases we are now permitted. While all sorts of fanciful rules could be dreamed up, the only one which is sensible (and has achieved a significant following) is to capitalize each distinct word. To give a successful example, the same procedure name becomes *EnsureExpressionValueIsOnStack*. This is certainly better. Its only disadvantages are that it is still somewhat unnatural, and that a desire for consistency often tempts programmers to capitalize identifiers of my classes (a) and (b) also. There is a real dilemma here: consistency vs readability—and no way of resolving it. Personally, I avoid the use of this device unless it is necessary, so that I should leave *symboltype* like that, but use it in procedure names like the earlier example. The capitalizing practice has been around for some time, and has been used in BCPL and other languages.

However, English-speaking programmers (including Americans, and Tasmanians amongst others) seem to prefer alternative mechanisms. I can only offer anecdotal evidence, in the form of an experiment with students. When our Pascal compiler was provided with the PL/I form of agglutination, but not publicized or promoted, a large number of students discovered it and its use spread like wildfire despite several disadvantages relating to the system print devices. If someone can do a carefully controlled experiment, here is a very interesting research question.

So, I ask myself, how can we best introduce English-style agglutination into new programming languages ? I think there are three reasonable choices:

(a) *Use the minus as a hyphen*
Since minus-signs are also used as operators, this option requires that we make the minus-operator lexically distinct, perhaps by requiring separating spaces in cases of doubt. This has a consistency disadvantage, for the plus-operator is not affected.

(b) *Use the underlined space as a hyphen*
This is the option used in PL/I, presumably because underlines are in the EBCDIC character set. In practice it suffers from a number of practical disadvantages. Some print and display devices have an awful underline that in fact runs through the bottom of the character; thus no-one can tell if *L* is underlined or not! Also, if the lines are closely-spaced, the underlines tend to merge with the next line. Typesetters, too, hate underlines.

(c) *Use the tilde as a hyphen*
No language I know uses this option, yet is is very attractive. The tilde is hyphen-like, even in English usage, it is a standard character in the ASCII set, and likely to be as widely available as lower-case letters.

There are other options which can be found in use in various corners of our computing world; for example the Burroughs file names use the slash (solidus) as an agglutinator. Examples of all these systems are given below.

SYSTEM	EXAMPLES	
Germanic	pixelarray	finddistinctionnode
Modified Germanic	PixelArray	FindDistinctionNode
Hyphenated	pixel-array	find-distinction-node
Tilde	pixel ~ array	find ~ distinction ~ node
PL/I	pixel_array	find_distinction_node
Burroughs convention	pixel/array	find/distinction/node

Personally, I look forward to the day when the tilde is made to work for its place in the ASCII character set. I regret that Pascal never allowed it in identifier names, for the resulting text is very easy to read and is free of the distractions of the Modified Germanic style.

The hyphenated styles do raise one new problem: should a hyphenated identifier be the same as an unhyphenated Germanic style one ? Is *x move* the same as *xmove* ? My answer is no, for the simple reason that word delimiting is meaningful. Thus *ranking ~ intact* means something different to me from *ranking ~ in ~ tact*, and from *rank ~ in ~ gin ~ tact* (though I would be puzzled by the last!). There are no prizes for finding other ways of breaking *rankingintact* into word sequences.

FREE STYLE

Most people would probably agree that identifiers should always be displayed in the same way, and would probably extend this to keywords too. Yet is there a case for treating some keywords differently from others, or in a context dependent manner ? In fact, some programmers have adopted emphatic styles which mark significant places in the program text by capitals. For example, the keywords *program*, *procedure* and *function* mark distinct units for which it is often necessary to scan. Within a program unit, *label*, *const*, *type*, *var*, and the first *begin* of the executable statement part begin significant sections, and *end* closes the unit. In this style these keywords are written entirely in upper-case letters, so that they stand out by contrast. For example:

```
FUNCTION power(x : real; i : integer);
  VAR
    xtopoweroftwo,      {x raised to powers of two}
    result : real;      {formulates result}
    reducedi : integer; {residual powers not in result}
  BEGIN
    reducedi := i;
    result := 1.0;
    xtopoweroftwo := x;
    {Establishes the invariant R1 trivially.}
    . . .
  END;
```

Note that because *VAR* is capitalized in this position is no argument for doing the same with *var* in a parameter list. Personally, though I concede this argument, I do not do it myself as I consider that the visual cues of indentation and blank lines are much more successful and quite adequate.

The other practice I have seen is a treatment of statements as though they were sentences, resulting in the capitalization of the leading character of each statement (if it begins with a reserved word). For example:

```
For i := 1 to nmax do
  Begin
    With table[i] do
      If key = value then
        . . .
```

Again, this does not do much for me, mostly because programming languages are much more hierarchical than natural languages. The 'sentences' within 'sentences' seem slightly awkward, but not so much that I object to the practice. Perhaps this awkwardness is a consequence of lack of familiarity; stylistic questions often get bogged down in arguments from habit.

CONCLUSIONS

Though we know a lot more about programming than we did in the days of the birth of Fortran, we have not come all the way yet. Even Pascal's lexical representation is not perfect, and the draft Pascal Standard has appeared with very poor stylistics.

What we all need to realize, and put into practice, is that the readability of programs is important. More important than minor implementation details or slight losses in compiler efficiency, and certainly worth thinking about rather than mindlessly copying traditions from the past.

REFERENCES

1. A. Addyman *et al.*, 'A draft description of Pascal', *Pascal News*, No. 14, 4–54 (January 1979), and *Software—Practice and Experience*, **9**, (5) 381–424 (1979).
2. A. H. J. Sale, 'Pascal stylistics and reserved words', *Software—Practice and Experience*, **9** (10), 821–825 (1979).