INTRODUCTION

Object Pascal programs are often difficult to read and the language, difficult to learn, because there are no standard style guidelines for encoding structural elements of the dialect. Furthermore, the structure of class constructs, control-flow, scope, indentation, inline documentation, and upper/lower case, etc., is not sufficiently constrained in the definition of the language. Structure and function are inextricably entwined. The free-form nature of Pascal, from which Object Pascal derives, engenders many of the problems that novices, in particular, have in mastering it, and experienced programmers have interpreting it. The solution is to map a set of coding conventions onto the language that impose the stylistic constraints needed to enhance program readability.

To enhance readability, the coding conventions described, here-in, are enforced by means of a structure editor we are developing at the New York Institute of Technology for Borland International’s Delphi, an Object Pascal-based integrated development environment (IDE). Delphi, according to the vendor’s promotional literature, owns a 40% share of the rapid application development (RAD) market for Wintel PC database applications.

Ironically, most of the computer science education community is now committed to transitioning from the use of Pascal to C++. Delphi’s success provides a strong argument to advance the use of Object Pascal (henceforth “Pascal”) in teaching. The argument is as follows: Pascal, long sited as a teaching language without industrial acceptance has, of late, found widespread adoption in industry. With this “second coming of Pascal” there is justification for preserving its use in CS1 and CS2 level courses, and further, imposing standards for program coding that are manifest in industry for other languages. In the absence of a structure editor, these may be imposed by instructor’s decree.

THE NEED FOR CODING CONVENTIONS

The bottom-line in structured programming practice is readability[2]. For a program to be considered readable, there must be an element of predictability to it. Becoming versed in a programming language entails gaining a sense for “where things go” in programs written in it. While experts have proficiency in the expression of programming logic at the semantic level, novices are encumbered with learning syntax[7]. Learning the placement of language elements is often treated merely as an exercise in logic. The issue of style is often, unfortunately, divorced from the issue of form. Questions such as, “What should these constructs look like,” and “Why so many variations in form?” arise naturally in the course of learning a language. Coding conventions, the programming equivalent of standardization of parts, effectively limit the number of stylistic variants. This is a small issue for expert programmers, but a very large one for novices.
No lore of structured programming style ever evolved with Pascal, because the coding conventions of a particular dialect were never adopted universally. This fault owes to the fact that no examples of a consistent coding style were presented in Jensen and Wirth’s original report on the language[5]. Furthermore, the ISO definition of the language appeared too late to stem the tide of dialects that flooded the market, once it became entrenched as the lingua franca of computer science education. Thus Pascal, ballyhooed as the archetype of structured programming languages, is often not used as a structured programming language in practice.

Coding conventions add the elements of predictability to Pascal programs that render them more readable. They are aesthetic conventions that constrain the coding style of the programmer. They are not part of the definition of the language, but are externally imposed upon it. Where the declarative and procedural elements of a Pascal program are placed is well defined in the language, and enforced by the compiler. The use of semicolons (;) and begin..end pairs as scope delimiters is quite flexible, however, and the programmer may code a variety of syntactic variants for a given program plan that are legal and functionally equivalent, but difficult to read. This is the tragic flaw of Pascal. The fact that many educators are unfamiliar with the use of coding conventions in industry has compounded the problem, and generations of their students have hit the market and will spend their careers writing horribly structured programs as the result.

Established programming languages such as FORTRAN and COBOL imposed certain constraints upon the coding style of the programmer because the limitations of their compilers forced them upon the programmer. Not more than one statement is permitted per line in these languages, and these must fall within the bounds of certain column positions in the source file. The aftermath of Dijkstra’s manifesto[3] on the use of goto statements in 1968 led to the introduction of new control constructs in these programming languages, so that FORTRAN by 1977 and COBOL by 1985 could be deemed "structured programming languages." The argument that gotoless programming is structured programming is inadequate, however. A program that uses gotos in a tempered and predictable manner can be designated "structured"[12]. The addition of DO, DO..WHILE, WHILE..DO, IF..THEN..ELSE like constructs to both FORTRAN-77 and COBOL-85 enhanced the readability of programs written in these languages. To his credit, they were recognized by Wirth and implemented in the definition of Pascal as early as 1970. In fact, they had been implemented in Algol by 1960, and introduced along with PL/1 in 1966[12]. Unfortunately, Algol suffered from many deficiencies, and early PL/1 compilers were riddled with errors, so that it was not until Wirth’s invention of Pascal that the concept of using these fundamental control elements was finally bought wholesale in North America.

A subtle but fundamental realization that came about as a result of the structured programming movement in the late 1960s was that a group of one or more adjacent statements in a language normally served to carry out a unified processing task. Furthermore, this group of logically related statements, deemed a block, was executed either unconditionally, as a simple sequence, or conditionally, subject to a logical test. In Pascal, the semicolon was adopted from Algol as a statement delimiter, and the begin..end pair as block delimiters. The semicolon can also serve as a statement terminator, however, and many evils of the language arise as the result of this added flexibility.

Wirth's intention was to design a programming language that would support structured programming and the composition of flexible data structures, yet be compact enough to run on a small computer. Pascal is the incarnation of an algorithmic language whose syntax facilitates the process of compilation. A statement in the language, be it a single statement terminated by a semicolon, or a compound of statements within a begin..end pair, is the fundamental unit of
compilation. While a statement is readily identifiable from the standpoint of the compiler, however, it is the responsibility of the programmer to format statements so that their scope can be discerned readily by visual inspection. Unfortunately, this aspect of programming is not emphasized as much as syntax or algorithm development, so that novice and professional programmers alike are free to layout their code segments in any form they like.

The grammars of FORTRAN-77, COBOL-85, and even modern dialects of BASIC[8], require the use of DO..ENDDO, WHILE..ENDWHILE, and REPEAT..ENDRPEAT like delimiters which force statements encapsulated within them to be coded as a visually discernible block. This is not true of Pascal. The three counter loops below contrast the differences between FORTRAN and Pascal in terms of readability.

**FORTRAN:**

```
I = 1
WHILE I .LE. 10 DO
   I = I + 1
ENDWHILE
```

```
I = 0
DO 5 I = 1,10,1
   I = I + 1
5 CONTINUE
```

**Pascal:**

```
I := 1;
while i <= 10 do
   i := i + 1;
until i = 10;
```

Notice that the semicolon serves as both a statement separator and as a statement terminator in the Pascal while-loop. In FORTRAN, a statement is delimited simply by virtue of its being coded on a line by itself, and the ENDWHILE statement serves the separate function of delimiting the block. The repeat statement is a little more tame, but the statement separator/delimiter problem arises once again in the case of the for-loop. Indentation helps in both languages to indicate scope of operations, but the dual purpose of the semicolon forces us to search for it in the while-loop and for-loop cases to find out where these statements terminate.

In these examples, the problem seems trivial, but this is not so when we consider that the syntactic variants in Pascal that follow are also legal and equivalent to their counterparts.

```
i := 1;
while i <= 10 do begin
   i := i + 1;
end;
```

```
for i := 1 to 10 do begin
   ;
end;
```

```
i := 1;
while i <= 10 do begin
   i := i + 1
end;
```

```
for i := 1 to 10 do begin
   end;
```

Where and when to use a semicolon in Pascal is a vexing problem for beginners[13, 14] and the flexibility of the language allows those with experience to adopt their own style. One common variant of the while-loop follows:

```
i := 1;
while i <= 10 do
begin
   i := i + 1
end;
```
The thinking here generally goes that since the block is executed within the scope of the while, it is "best" to indent it under the while. Unfortunately, the reader of this code segment must carefully inspect it to find the delimiting end statement and terminating semicolon, and this becomes even more difficult if the compound of statements within the block spans more than one page or screen.

The problem grows worse in nested structures, especially when there is a bug[10]! For instance, examine the following program segment:

```pascal
for --- do
  ---;
  while --- do
    begin
      ---;
      ---;
    end;
```

The for-statement is terminated, prematurely in this case, though the indentation would lead us to believe that the while was to be executed within the scope of the for. This problem has caused many to disdain the use of Pascal, which the late Alan Perlis of Yale once diagnosed as suffering from "cancer of the semicolon."

**THE ART OF WRITING READABLE PASCAL**

The solution to this problem is simply to require that both simple and compound statements be treated as blocks that are encapsulated within a begin..end pair, and to place the end keyword delimiter under the reserved word that begins the block. We can then use semicolons wherever we like (except before else statements) since they serve simply as statement separators in that case, the function of delimitation having been off-loaded to the end statement. Strategically placing the ends directly under the reserved words they delimit, and putting the begin statements off to the side as we have done in the above examples, allows us to visually inspect a statement block and determine precisely where it begins, where it ends, and which statements are subsumed by it. The additional advantage of this approach is that it simplifies the syntax of the language for the novice, making it easier to learn with fewer rules and variants to remember[14], and constrains programs to visual forms which easier to comprehend. Thus our example would be rendered more readable if coded as follows. Notice that our earlier "error" becomes a non-error and that, in fact, had we left out any semicolons using this style, the compiler would have flagged the essential cases!

```pascal
for --- do begin
  ---;
  while --- do begin
    ---;
    ---;
  end;
end;
```
If-statements in Pascal are even worse! Six syntactic variants are illustrated below.

\[
\begin{align*}
\text{if} \quad &\quad \text{then} \\
\quad &\quad \text{begin} \\
\quad &\quad \quad \text{else} \\
\quad &\quad \quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then} \\
\quad &\quad \text{begin} \\
\quad &\quad \quad : \\
\quad &\quad \quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then} \\
\quad &\quad \quad \text{else} \\
\quad &\quad \quad \quad \text{begin} \\
\quad &\quad \quad \quad \quad : \\
\quad &\quad \quad \quad \quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then} \\
\quad &\quad \text{begin} \\
\quad &\quad \quad : \\
\quad &\quad \quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then} \\
\quad &\quad : \\
\text{if} \quad &\quad \quad \text{begin} \\
\quad &\quad \text{begin} \\
\quad &\quad \text{end}; \\
\end{align*}
\]

The six forms above can be reduced to the two forms that follow.

\[
\begin{align*}
\text{if} \quad &\quad \text{then \hspace{0.5em} begin} \\
\quad &\quad \text{begin} \\
\quad &\quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then \hspace{0.5em} begin} \\
\quad &\quad \text{begin} \\
\quad &\quad \quad \text{end}; \\
\end{align*}
\]

Nested ifs can then be handled just as easily. A preponderance of end-statements usually comes at the bottom of such structures, but these serve as rule marks along which students can lay a straight edge and trace a line to matching ifs or elses to determine the scope of a particular condition. The following example illustrates this point.

\[
\begin{align*}
\text{if} \quad &\quad \text{then \hspace{0.5em} begin} \\
\quad &\quad \text{end}; \\
\quad &\quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then \hspace{0.5em} begin} \\
\quad &\quad \text{end}; \\
\quad &\quad \quad \text{end}; \\
\text{else \hspace{0.5em} begin} \\
\quad &\quad \text{end}; \\
\quad &\quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then \hspace{0.5em} begin} \\
\quad &\quad \text{end}; \\
\quad &\quad \quad \text{end}; \\
\text{if} \quad &\quad \text{then \hspace{0.5em} begin} \\
\quad &\quad \text{end}; \\
\quad &\quad \quad \text{end}; \\
\text{else \hspace{0.5em} begin} \\
\quad &\quad \text{end}; \\
\quad &\quad \quad \text{end}; \\
\text{end}; \\
\end{align*}
\]
The approach to style mandated here evolved after numerous trials with novice and intermediate programmers at several universities. Not only did the most common sorts of scope errors disappear, but students were able to concentrate more on learning algorithmic concepts. Moreover, numerous studies have lent support to these observations. Sykes, Tillman, and Shneiderman found strong evidence for the value of scope delimiters, such as `begin..end`, in increasing program comprehension[14]. Miara, Musselman, and Shneiderman also studied the effect of indentation on program comprehension, and found that utilizing a three space indent to denote the scope of operations was optimal when using non-proportional fonts[10]. Note that alternatives to the approach advocated in this paper introduce additional levels of indent that are not optimal. Soloway, Bonar, and Ehrlich[13] argue for the use of keyword scope delimiters, and further suggest that the conditional test for exit be encapsulated inside the block of statements constituting a loop as shown below.

```
WHILE-DO loop:
  loop
    if --- then quit;
  : endloop;

DO-WHILE loop:
  loop
    if --- then quit;
  : endloop;
```

The use of lower case in these examples is a convention suggested by Dennis Ritchie in the original implementation of the "C" programming language, and further advocated for Pascal by Kernighan and Plauger[9]. Fewer typos are committed as the result of keystroke errors when lower case, the default option on most keyboards, is used. The idea here is that fewer dual key combinations, requiring shift-key pairs, leads to fewer errors. There is also strong psychological evidence that lower case letters are easier to read, since their more varied forms are easier to discriminate[1]. Kernighan and Plauger advocate restricting the use of upper case to filenames, buffer and device names, and constants. Thus in the following examples it is evident by inspection that PI is a constant (3.14159) and that STUDENT is a file buffer variable and not a field! The latter would not have been at all apparent were STUDENT coded in lower-case as student, unless we read the program from start to finish.

```
area := PI * sqr (radius);
readln (STUDENT, grade);
```

Phil Miller of Carnegie Mellon has argued that many of the structural faults of Pascal, pointed out above, can be reduced if students use a structure editor which generates templates of syntactic structures automatically, allowing students to take a "fill-in-the-blank" approach to coding (personal conversation). His Genie Integrated Pascal Development Environment on the Macintosh works in this fashion. He agrees that there are probably better alternatives to Pascal as a "first" programming language, and that Pascal's successor, Modula-2, has a cleaner syntax that avoids all of the problems pointed out above. Block structure is enforced implicitly in Modula-2, and all keyword control statements must be delimited by matching ends, as the examples that follow illustrate.

```
if --- then
  while --- do
    for --- do
      repeat
        ---;
    : end;
else
  ---;
end;
```

61
Notice that the keyword delimiter `begin` is no longer required in Modula-2 to signal the start of a block. Because of this cleaner syntax, a colleague noted that when he rewrote his bestselling Pascal text in Modula-2, the examples ended up taking considerably less space, and learning was greatly facilitated[6]. Wirth himself abandoned Pascal by his own accounts, and strongly advocated Modula-2 for a time. Old habits die hard, however, and Modula-2 never gained a foothold in either academia or the marketplace.

**DOCUMENTATION, DECLARATIONS, HEADERS, AND CLASS CONSTRUCTS**

The basic design principles described above are extended to other program elements. For inline documentation, we appropriated the "manual page" conventions advocated by Kernighan and Plauger[9], with some minor modifications. There are two sets of tokens used to delimit comments in Pascal, (* and *), or { and }. Only curly brackets {}, are permitted in our editor for this purpose. Furthermore, while Pascal would permit comments to be interspersed throughout the code, even at the end of a line of executable code, we restrict the use of comments to blocks after the main **program**, **procedure**, or **function** headers only. The idea here is that subprogram size should be restricted to a page or so of self-documenting code, and comment blocks are useful only in describing generalities about the program or subprogram. The following template illustrates the components of a documentation block.

```plaintext
{

FILE : ANYFILE.PAS
MODULE : Descriptive program name
AUTHOR : Last name, First name
REVISED : MM/DD/YYYY

DESCRIPTION : A terse, multiline description of what the program or subprogram does.

BUGS : A terse, multiline description of known bugs or limitations of the program or subprogram.
}

program or subprogram header goes here..
```

Constant, type, and variable declarations follow the case conventions described earlier, and follow the program or subprogram header. Only one declaration is permitted per line, following the spacing conventions of the documentation block:

```plaintext
const
  TAX = 0.08;
  SIZE = 80;

type
  name_type : string [SIZE];

var
  cost : real;
  price : real;
```

Program and subprogram headers contain parameters on one line as illustrated below.

```plaintext
program Grade_calculation (INPUT, OUTPUT);
function average (var score: reals; n: integer): real;
```
The thinking here is that any program or subprogram header that contains more than one line’s worth of parameters ought to be simplified, or decomposed into subordinate routines. Note that no space is permitted before punctuation marks, while one space is required after, just as in written natural languages. The arrangement of the parameters is further intended to mimic the functional notation used in mathematics:

\[ y := f(x, y, z); \]

Note that arithmetic operators require one space before and after for readability.

Finally, class declarations incorporate all of the executable elements described above. Their general form is patterned after type declarations, of which they are a subset:

```plaintext
type
class_type = class (parameter_list_1);
x : type_1;
y : type_2;
z : type_3;

procedure first (parameter_list_2);
procedure second (parameter_list_3);

private
{
    Private declarations
}
public
{
    Public declarations
}
end;
```

CONCLUSION

Several measures of the efficacy of the coding conventions described in this paper have been obtained. Performance-based measures indicate that more students are able to complete assignments in less time. Moreover, collaborative learning is enhanced in lab settings, as a greater sense for structure and style enables peer programmers to assist one another with greater ease. Apparently, it is easier to spot malformed or missing plans in a buggy program. Style conventions impose a set of expectations for a program design: students report a sense of unease when an implementation does not match their expectations.

Analytic grading of programming assignments is also facilitated as the criteria for correct designs are clearly spelled out. We have observed that the majority of program errors tend to be structural. In effect, we have proscribed a set of templates for correct designs that should match the students’ metacognitive models for proper constructs. All the standard algorithms for CS1 and CS2 are thus presented in a style conforming to our guidelines.

Whether Object Pascal remains in favor or loses out to C++ in academia and the market place, style guidelines are of merit in any programming context. The rapid ascendance of Java in the programming world suggests that we can expect radical evolution of language features and application paradigms. In the meantime, we can better prepare our students for the inevitable transitions by teaching and continually emphasizing the rudiments of programming style.
REFERENCES


