Programmer Experience-Level Indicators

J. A. W. FAIDHI AND S. K. ROBINSON
Department of Computer Science, Brunel University, Uxbridge, Middlesex UB8 3PH

If we interpret 'teaching' as a process of controlled learning it is tempting to try to mechanise the teaching of programming as a feedback control process. One major difficulty is the identification of program measures that indicate student experience which can be automatically collected. Based on a static empirical analysis of student Pascal programs, we detail those measures that we have found to be affected by programmer experience. We also give the general results of the empirical analysis which are of interest to other researchers.

Received October 1985, revised July 1986

1. INTRODUCTION

'Teaching' can be defined as a process of controlled learning,[28,6] and from this it has been suggested that the process of teaching could be mechanised as a control process or, more specifically, as a series of prescriptions provided for the controlled subject (the student). For reasonable control, a prescription must often be contingent upon evidence of the current state of the controlled subject.

It is generally accepted that any course which attempts to teach computer programming as a control process must require its students to actually write programs and to run these programs on a computer. It is helpful for those teaching the course, the 'controllers', to monitor the source code and the run-time results, as this provides *feedback* which can be used in a number of ways to improve the learning process:[32]

(1) To provide the basis of a formal assessment of student performance.

(2) To indicate the part of the course that has been understood, and which parts are still unclear. This allows material to be re-emphasised.

(3) To show how well individual students are coping with the course, so that additional tuition can be provided for those who need it.

Unfortunately, there can be difficulties in obtaining feedback data automatically. Most difficult of all is finding metrics appropriate for the analysis of student progress. Researchers have used two different approaches for this. One quantifies the behaviour and performance of the programmer him/herself,[29] whilst the other utilises models of complexity to assess the programs produced.[3]

Studies of programmer behaviour measure performance indirectly by scoring how well the subject can memorise, modify, debug, hand-trace and answer questions. It is assumed that the easier a program is to comprehend, the easier it is for the programmer to accomplish the task. However, many researchers have found difficulties in attempting to mechanise program understanding, both at the semantic and the pragmatic level.[30,31]

Program complexity models try to identify factors that contribute to the complexity of software and thereby derive objective metrics. The depth of analysis required to obtain any metric, and the relevance of the source language involved, varies between models. The software science model,[5,2] for example, simply requires the identification of operators and operands, whereas models based on the empirical approach[3,19,25] require specific knowledge of the syntax of the source language as well as knowledge of the *baseline* sample. The empirical approach offers many applications, particularly in a teaching environment: it can indicate plagiarism between programs[7,25] and it can aid in the construction of automatic marking schemes for student assignments[23,26] (including attempts to combine this with improvements in the learning process).[6] The empirical approach yields results that prove useful in other areas including code optimisation,[3,8] the construction of empirical prediction formulae,[1,5] computer architecture quality evaluation[4,24] and language efficiency monitoring.[27]

2. THE SAMPLE SOURCE LANGUAGE

Within a university teaching environment many programming languages are used. While certain languages are considered unsuitable for honours computer science

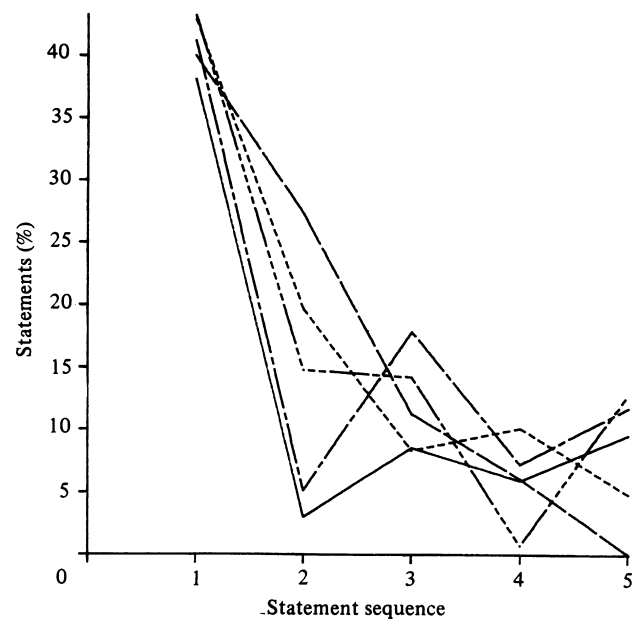


Fig. 1. Comparison of Pascal with other languages. ———, Pascal; – – – –, Algol; ——, FORTRAN; ———— —, COBOL; ——— — —, PL/1. Statement sequence: 1, Assignment; 2, Procedure call; 3, Conditional; 4, Loop; 5, GOTO.

(such as BASIC),[9] others, Pascal for example, have distinct educational advantages. Fig. 1 empirically compares a Pascal language sample with other language samples, all produced by university students. We see that, in general, Pascal programs use procedure/function calls more frequently and contain less GOTO statements. Even this simple observation suggests that Pascal is a better teaching vehicle. Pascal, when introduced, "was a significant contribution to the evolution of programming languages. It is a simple but powerful language in the tradition of Algol 60.[11] Compared with Algol 60 it contains more mechanisms for the definition of data structures, enumeration types, records, pointers, and input/output."[10] Pascal is the "perfect" teaching language.[6] Compared with FORTRAN, besides containing better data structures and access mechanisms, it has been shown that a Pascal program is substantially faster in both compilation and execution speed than its equivalent FORTRAN program.[12] For these reasons, and others, Pascal is one of the languages utilised at undergraduate level at Brunel University, and as such was used as our sample source language.

3. THE EMPIRICAL APPROACH

For an empirical approach to complexity measurement two kinds of statistical analysis may be used:

(1) A static analysis, which records occurrences of various features of a source program sample. The results of this analysis reflect how a language is used in actual program source.

(2) A dynamic analysis, which investigates program execution. This may be used, for example, to study the frequency of specific types of expression in execution, or the frequency of reference to a declared name or constant.

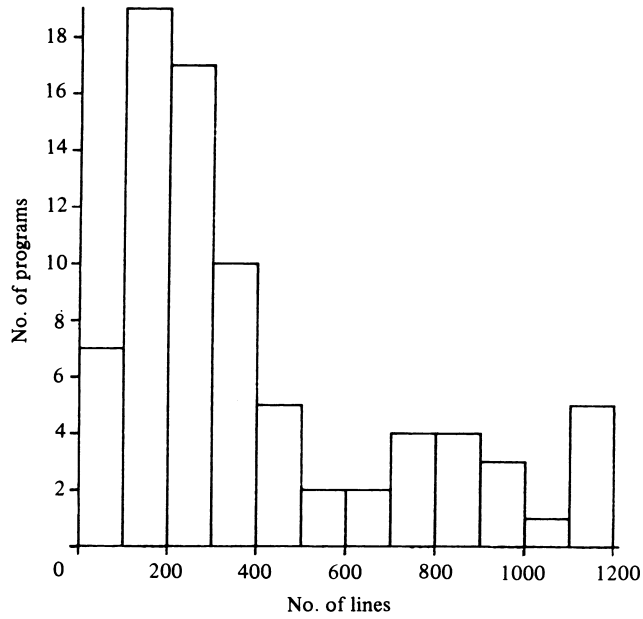
Both types of analysis for Pascal have been reported by Shimasaki *et al.* (1980)[13] and Brooks *et al.* (1982).[14] However, compiler source was used as the analysed data in the first study whilst a comparison study between scientific and non-scientific programs was undertaken in the second. Further levels of detail in a Pascal static analysis are presented here.

As we require to identify those program complexity metrics that reflect experience, our analysis ranges over three student program samples that embody the undergraduate experience spectrum. There were four potential samples, one for each year of the computer science undergraduate course at Brunel. However, the third year at Brunel involves little in Pascal programming coursework, and hence was omitted from the study.

4. TOWARDS A COMPLEXITY MEASUREMENT OF PASCAL PROGRAMS

The result of applying a static analysis on programs is the production of a static profile. The main mechanism of static analysis is the counting of the occurrence of certain program features. For our analysis the questions to be answered, then, are

- What features of a program contribute to its complexity?
- How do we count these features within the program structure?



	Year		
	1	2	4
Program source length (lines)	No of programs according to their length		
0–100	4	2	1
101–200	18	0	1
201–300	14	2	1
301–400	6	2	2
401–500	2	1	2
501–600	0	2	0
601–700	1	1	0
701–800	0	4	0
801–900	1	3	0
901–1000	1	2	0
1001–1100	0	0	1
1101+	0	0	5

Fig. 2. The University sample distribution

Too often, approaches to complexity measurement centre on a particular aspect of a program without incorporating other relevant program characteristics. Here we find the Delphi Survey,[15] on program design, constructive in that it identifies those essential features that affect complexity.

Counting the occurrences of program features is not trivial in the case of Pascal due to the nested structure of statements.[13] To achieve a systematic solution to this problem two approaches are possible: to produce a specific syntax analyser; or to modify an existing compiler. To achieve flexibility in our counting method we chose the first approach.

To achieve results all that needs to be done is to run the syntax analyser on source samples. Fig. 2 shows the distribution of student programs within our samples. Each stage sample group was arranged to have approximately the same size (average 10,900 lines). Seventy-nine programs in total were analysed. The longest program in this analysis had 2,354 Pascal lines. The sample consists of 47 programs from the first year, 19 programs from the second year and 13 programs from the fourth year.

Table 1: Distribution of basic Pascal statements (where sys pr-call indicates system, predefined procedure call, and user pr-call indicates user-defined procedure call).

Statement	First year (%)	Second year (%)	Fourth year (%)
Assignment	39.45	43.61	36.67
sys pr-call	28.16	16.52	13.81
compound	12.32	16.62	20.72
if	11.82	11.89	13.85
repeat	3.19	1.11	0.14
for	2.31	2.48	1.68
while	1.56	1.85	2.23
case	0.63	0.63	0.35
user pr-call	0.52	5.23	9.80
with	0.00	0.02	0.40
goto	0.00	0.01	0.24

5. THE GENERAL CHARACTERISTICS OF UNIVERSITY UNDERGRADUATE PASCAL PROGRAMS

In this section an analysis of the general pattern of the language use for a university undergraduate environment is reported. The results are given in relation to four attributes that can be seen to be related to the experience level of the student:

- Pascal statement utilisation,
- declaration profile,
- readability features,
- other programming elements.

5.1 Basic statement profile

Each successful statement recognition was counted; the results are displayed in Table 1. As can be seen, several statement utilisations vary with the experience level, most noticeably (*a*) the compound, if, while, user-defined procedure call and goto (positively), and (*b*) the system-defined procedure call and repeat (negatively).

The average distribution of statement types compares well with that from the study on scientific and non-scientific, large system programs performed by Brooks *et al.* (1980).[14] Fig. 3 summarises this comparison. The most utilised statement type proves to be the assignment statement, with an average 40% of the source. Standard procedure call statements come second highest, proving to be approximately 19% of a program. Compound statements (begin–end) and if statements come next highest, proving to be 17% and 13% of a program respectively. Among those having the lowest percentage are the 'case', 'with' and 'goto' statements, throughout the sample stages. Here we should note that the low utilisation of 'with' in the first year is due to difficulty in setting realistic elementary assignments that require its use while the low utilisation of the 'goto' statement is due to the lecturer's instruction that it should be used only when suitable.

5.2 Declaration part profile

The declaration components found to vary consistently (Table 2) with the experience-level of the students are (*a*) the array and the char type (negatively), and (*b*) the pointer and the subrange type (positively). Although the most utilised data structure is the array, the more experienced students tend to use others, such as pointers and records, instead.

The Pascal aggregate types which were most frequently employed (independent of experience) were the array (average 60%), record (20%) and pointer (9%) (all percentages are averages). The most significant Pascal simple types were integer (61%), real (3%), boolean (18%) and char (11%). The university undergraduate sample has an overall low utilisation of enumeration (3%) and subrange (average 3% of the total simple types declared). Although the ability of these features to improve the readability of source code is pointed out to

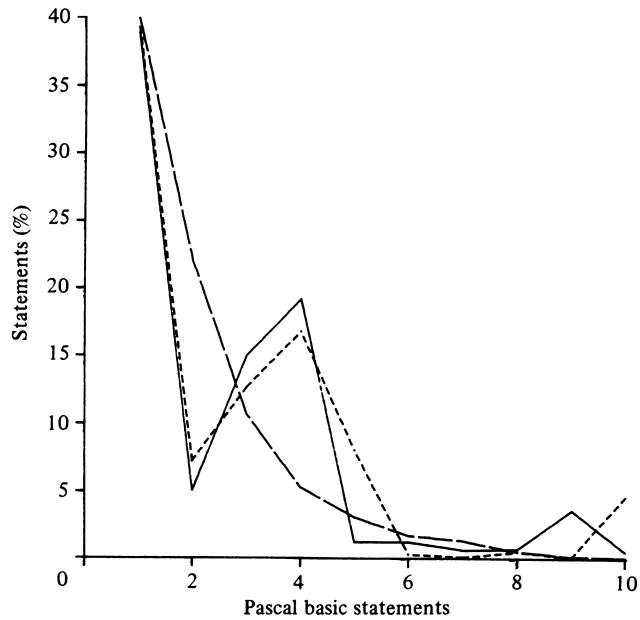


Fig. 3. Comparison of student programming sample vs. system samples. ———, Average student sample; – – – –, System sample (scientific); ——, System sample (non-scientific). Pascal basic statements. 1, Assignment; 2, Procedure call (standard); 3, If; 4, Procedure call (user); 5, For; 6, While; 7, Repeat; 8, Case; 9, With; 10, GOTO.

Table 2: The declaration part distribution in the university sample (excluding aggregate components)

	First year (%)	Second year (%)	Fourth year (%)
Aggregate type			
array	76.92	52.63	49.43
record	2.58	30.17	26.54
set	17.94	5.63	8.53
file	2.56	0.00	0.00
pointer	0.00	11.57	15.50
Simple type			
integer	52.13	66.18	65.08
real	2.00	6.14	2.00
boolean	19.07	16.32	19.12
subrange	1.00	2.62	4.11
enumeration	2.25	0.00	7.85
char	23.55	8.74	2.02

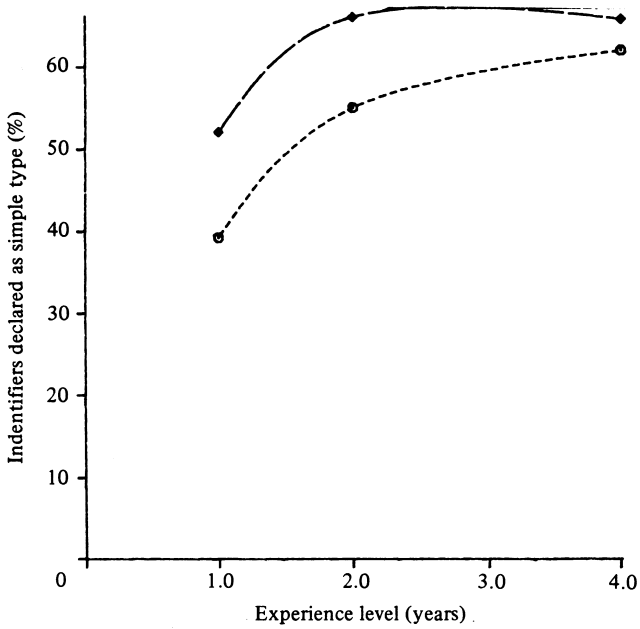


Fig. 4. The declaration of 'integer' names.
—◆—, Integer identifier; --○--, Integer 'predefined' word.

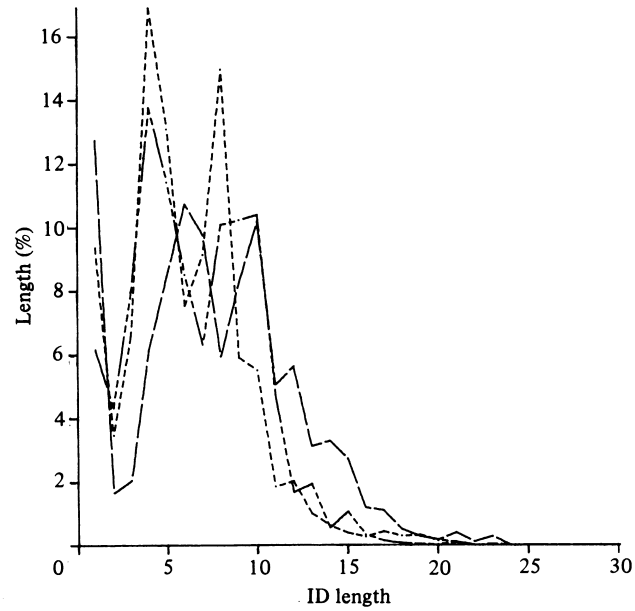


Fig. 6. University 'identifier length' distribution.
———, First year; – – – –, Second year; ----- Fourth year.

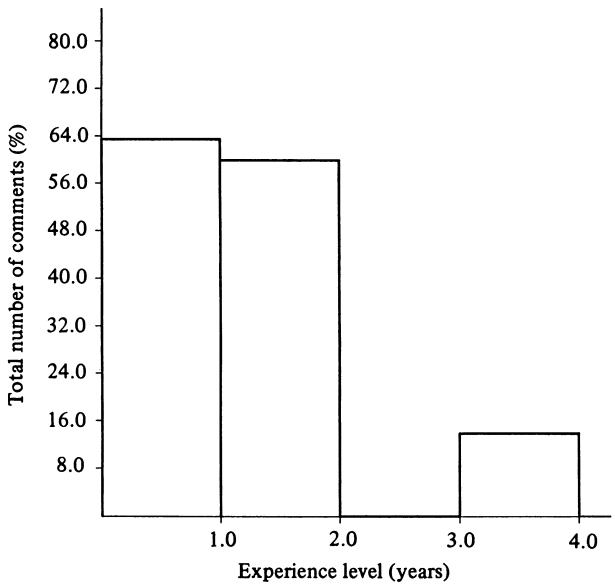


Fig. 5. Experience level vs. comments percentage.
□, University undergraduate sample.

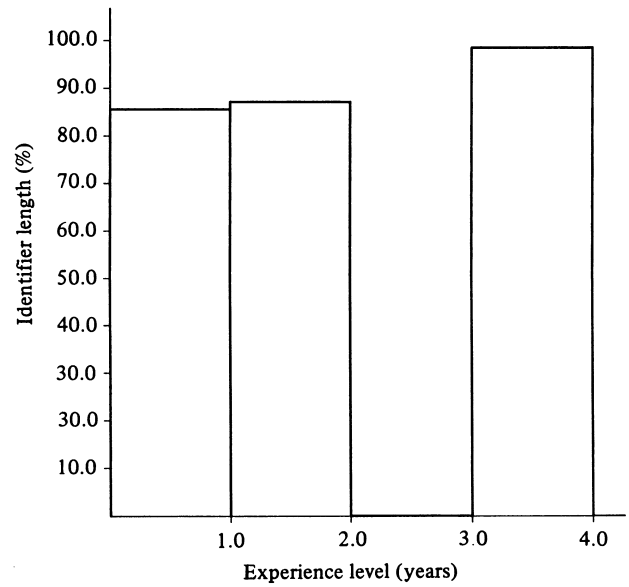


Fig. 7. Experience level vs. identifier length percentage.
□, ID length ⩾ 3 characters.

the undergraduate it appears to take some time for this to become apparent in practice.

It is interesting to note that we counted the number of names declared of a certain type rather than counting the type identifier 'integer'. Fig. 4, however, illustrates that this was unnecessary.

5.3 The program readability profile

The readability of a program cannot be automatically measured. However, specific attributes which affect readability can be identified.[16] The use of meaningful identifiers, suitable indentation, pagination and descriptive comments all improve readability.

The students at Brunel University are advised to use an indentation package 'pind', which is suppported by the Honeywell Level 68 (Multics) system, so we expect their programs to be indented correctly. Moreover, although comments may help to increase program readability, a simple count of the number of program comments can represent a misleading measure of readability.[17] Fig. 5 illustrates the percentage of comments used in our sample, and we find the first-year programs showing the highest usage! Other readability measures suggest themselves. We consider three: an identifier length measure, an average module size measure and a program modularity measure.

The identifier length may represent an effective measure for readability: the longer an identifier is, the more likely it is to be readable. Fig. 6 shows the distribution of identifier length (up to 30 characters) from our sample. We see that the use of more lengthy

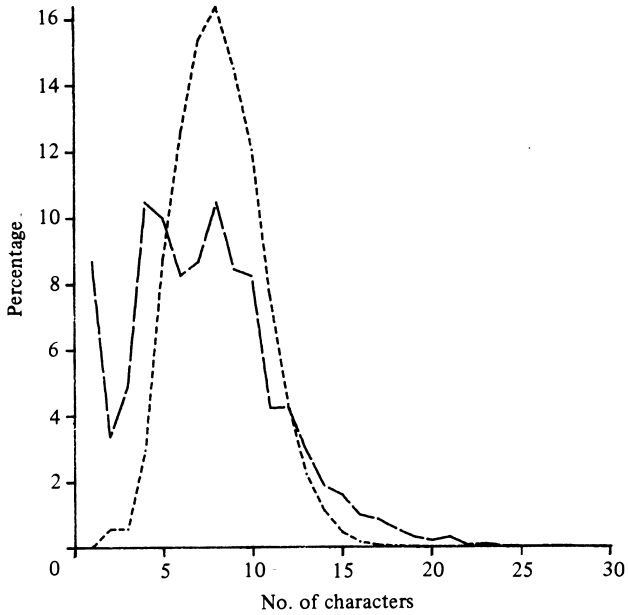


Fig. 8. Length comparison of Pascal identifiers and English words. ———, Average student Pascal identifier length (char); – – – –, English word length (char).

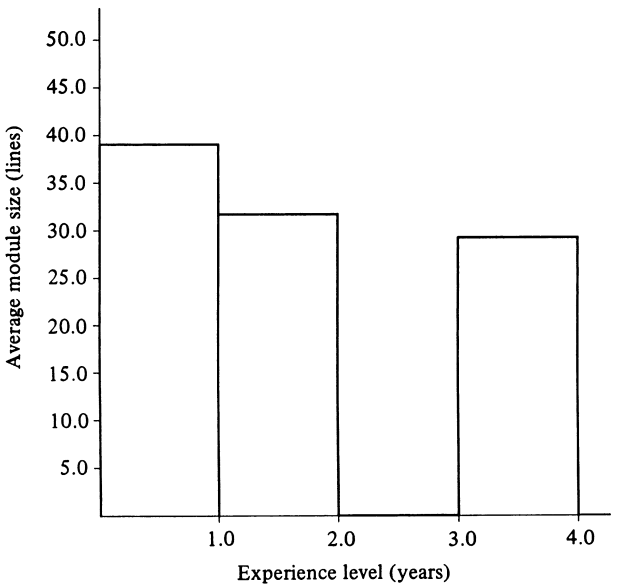


Fig. 9. Experience level vs. module size.
□, Function and procedure size.

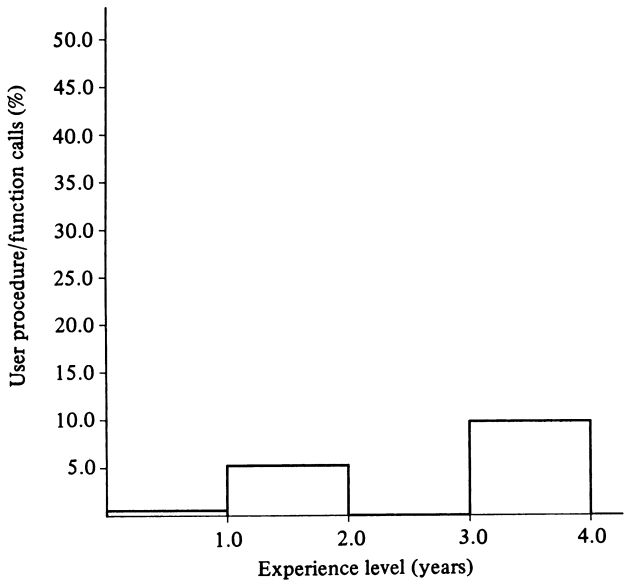


Fig. 10. Experience level vs. user module calls.
□, Percentage of user module calls.

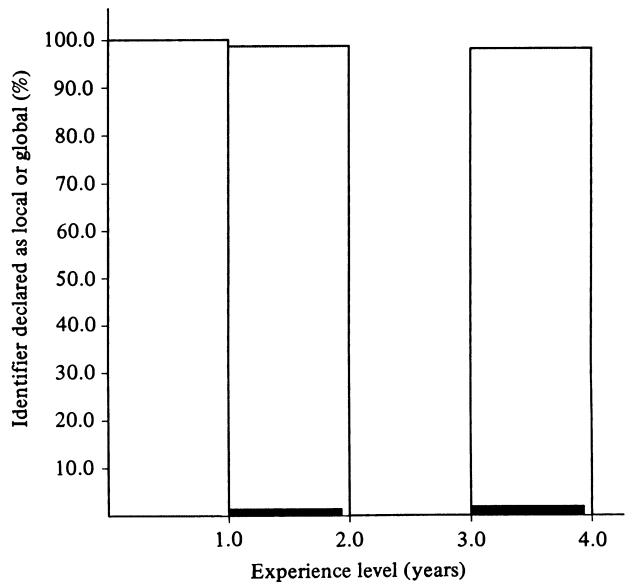


Fig. 11. The scope of declarations.
□, Local identifiers; ■, Global identifiers.

identifiers increases with experience as first-year students most frequently utilise identifiers of length one (13%), whereas second- and fourth-year students mostly utilise five characters (13%, 11%) or eight characters (15%, 10%) respectively. To clarify this point, Fig. 7 illustrates the use made of identifiers with three and more characters.

As an aside, a further result was obtained when Pascal identifier length was compared with the word length in a word processor's spelling checker's dictionary (Fig. 8). Here we find that the Pascal identifier length distribution is similar to that of English word length, suggesting that Pascal identifiers are close to English word structure. The only difference is that the students use slightly less identifiers between lengths 5 and 10 characters and slightly more between 1 and 5 (mostly caused by the first year's high use of single character identifiers).

The average module size (procedure-function) may be, in our opinion, another readability factor: the shorter a module is, the more readable it is likely to be. Fig. 9 shows the behaviour of this measure. Again it proves to be related to experience. The use of procedures and functions, as presented in Fig. 10, shows that program modularity also increases with an increase in experience. Using these three measures we find that first-year programs are not necessarily the most readable as was suggested by comment usage, but that, in general, the readability of programs increases with the experience level.

A detailed study that compares grades from experience-dependent style measures (including the readability

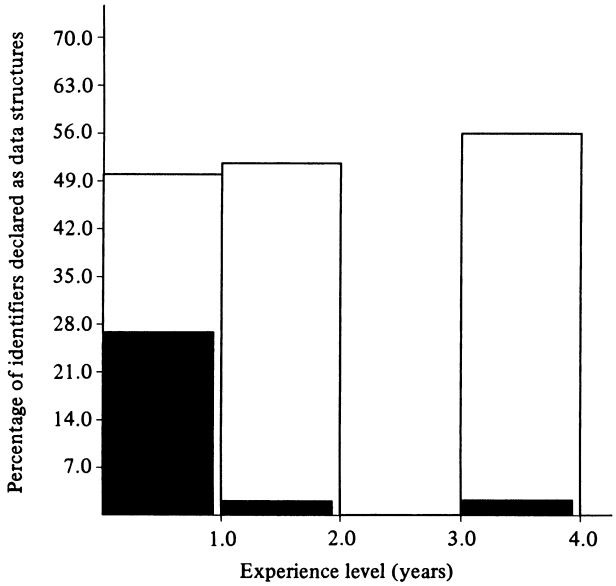


Fig. 12. The static utilisation of arrays.
□, Single-dimension array; ■, Multi-dimension array.

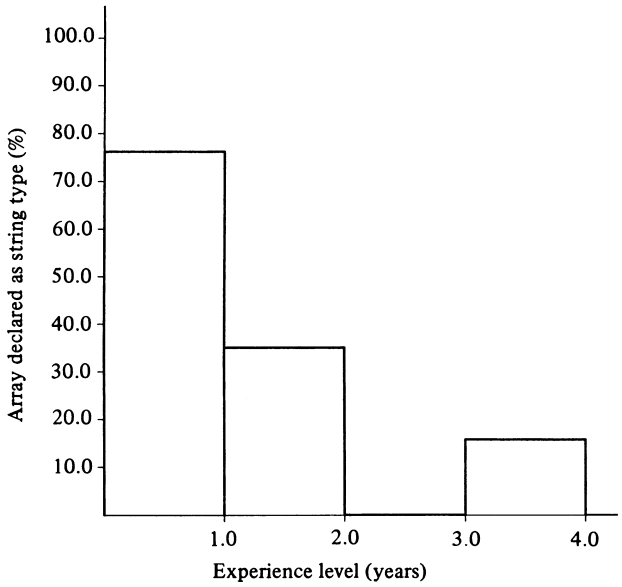


Fig. 14. String utilisation.
□, Percentage of the total number of arrays.

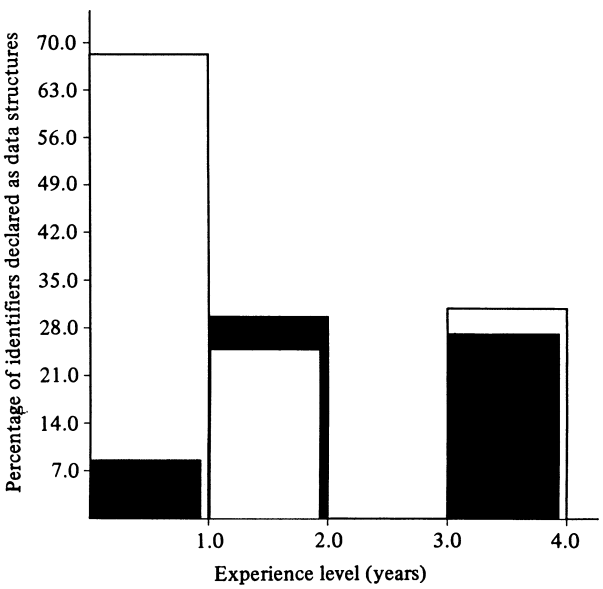


Fig. 13. The static utilisation of arrays.
■, Packed array type; □, Unpacked array type.

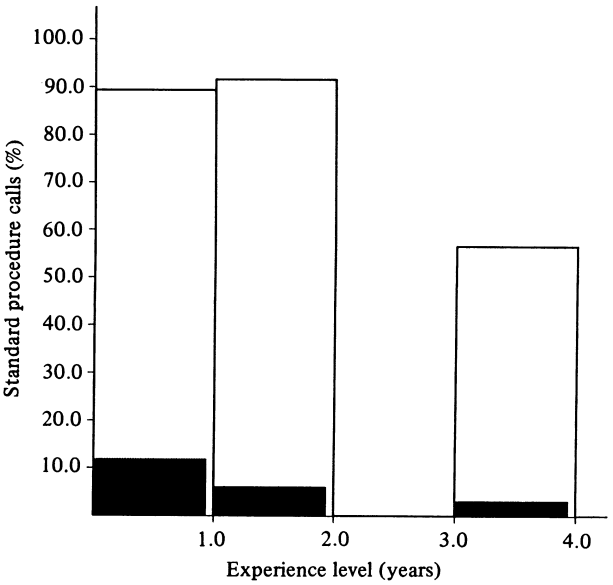


Fig. 15. The utilisation of I/O procedures.
■, Input procedures; □, Output procedures.

metrics) with those from human assessors is reported by the authors elsewhere.[26]

5.4 Other programming measures

Fig. 11 shows details of the two scopes that names may possess: those declared in the main program declaration part (global) or those declared in the declaration part of a block (local). It is known that global identifiers affect the program structure by increasing a module's coupling, whereas local identifiers exist only in the lifetime of the declaring block and hence do not affect block dependency (cohesion). We would suggest a preference for variable localisation over the use of global variables, as their use minimises certain program complexity measures.[22] We note that this preference is not supported by all researchers and that some advocate exactly the reverse.[21] We see, from Fig. 11, that although students in general use more local identifiers than global, a trend is visible towards the use of global variables as experience increases.

Since arrays are the most frequent data structures used in the sample (Table 2) it is interesting to detail their use further. The factors contributing to array size and access are packing and dimension. Fig. 12 illustrates the use made of single- and multi-dimensional arrays within the university sample. Single-dimension arrays are used in preference to multi-dimensional (though this may well reflect our setting of course-work), and the use of multi-dimensional arrays decreases with experience. Packing concerns the density of data in memory: it should decrease memory utilisation but may be slower to access. Fig. 13 illustrates the use of packed and unpacked arrays in the university sample. Since the most common use of

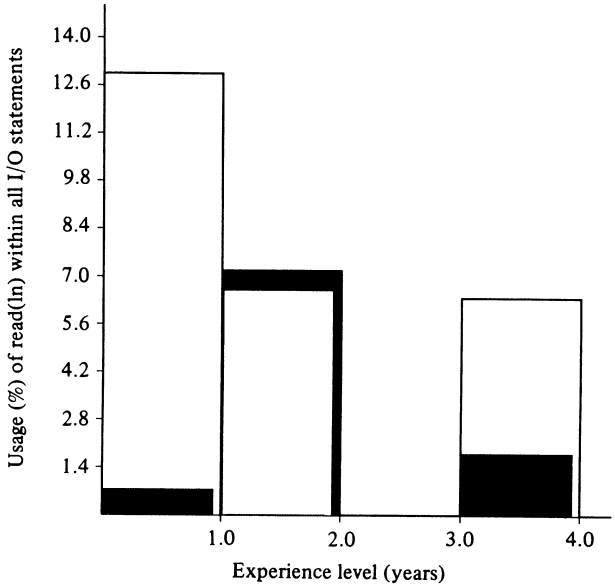


Fig. 16. The parameters of input statements. □, Read(ln) with one parameter; ■, Read(ln) with two parameters; ▥, Read(ln) with three parameters; ▤, Read(ln) with ⩾ 4 parameters.

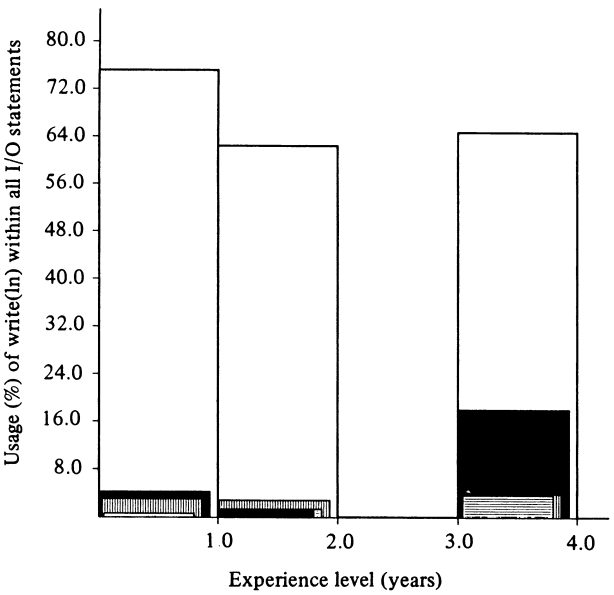


Fig. 17. The parameters of output statements. □, Write(ln) with one parameter; ■, Write(ln) with two parameters; ▥, Write(ln) with three parameters; ▤, Write(ln) with ⩾ 4 parameters.

packing is for defining 'strings' (packed array of char), Fig. 14 illustrates this use, and we note a decreasing trend in use with experience.

Finally, we look at the way university students utilise I/O routines. Their use is problem-dependent, but at the same time I/O plays an important role in program testing. To measure I/O use we counted the use of 'read(ln)' and 'write(ln)' statements in the sample and the number of variables that were read or written in each I/O statement ('get' and 'put' were also considered but were found not to occur). Fig. 15 illustrates the use of the I/O procedures among the standard procedure calls and we note a decreasing trend in the use of input procedures

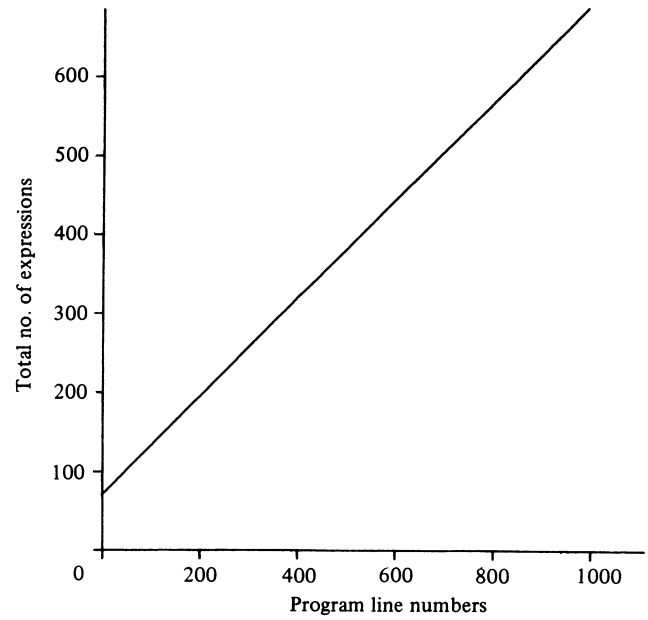


Fig. 18. The relation between number of expressions and number of lines. ——, Sample taken from second year.

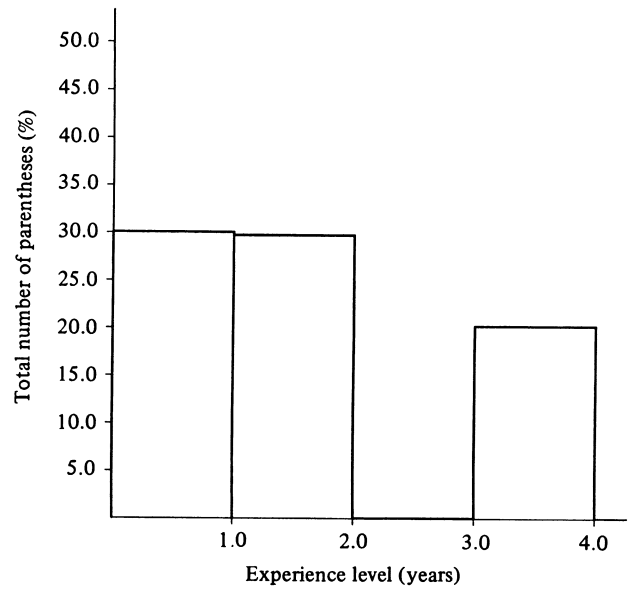


Fig. 19. Experience level vs. parentheses percentage. □, Percentage of parentheses or grouping factors.

with experience. Figs 16 and 17 detail the number of variables read and written respectively.

6. A DETAILED ANALYSIS OF PASCAL EXPRESSIONS

In this section we analyse expressions. Expressions play a very important role in program complexity.[18] There is an expression in nearly every program line, as Fig. 18 illustrates. The way expressions are utilised not only affects complexity but also alters readability and efficiency.

Conventionally, expression complexity is judged by the number of parentheses (grouping factors) utilised; the less they involve the less complex they are.[20] Fig. 19 illustrates the result of applying this technique to our

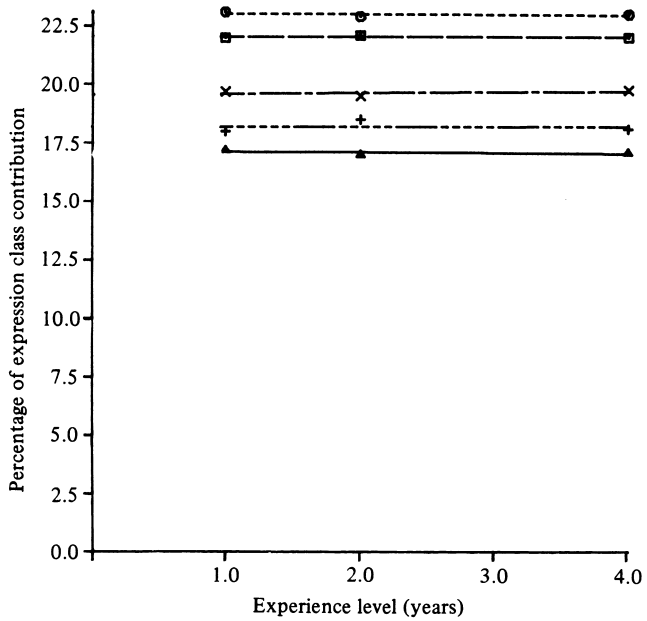


Fig. 20. The utilisation of expression components. —□—, Factor; --○--, Term; —▲—, Simple expression; —+—, Expression; —×—, Complex expression.

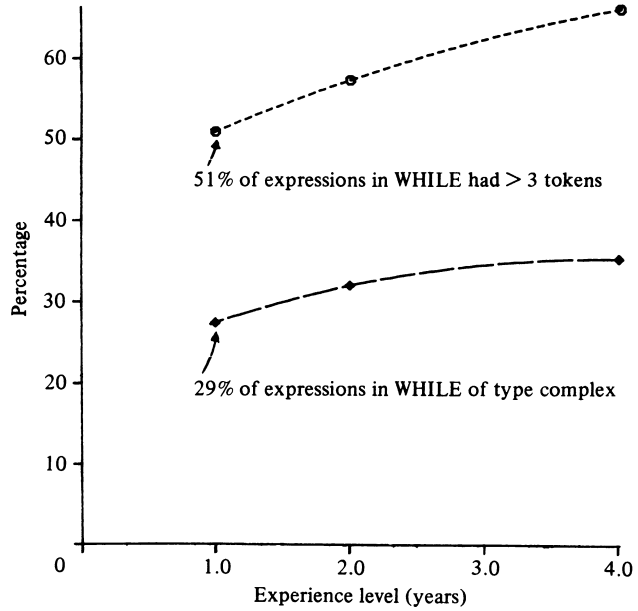


Fig. 21. Complex expressions within a 'while' statement. —◆—, Complex expression; --○--, Complex expression tokens.

Table 3: The relationship between the expression classes in each main Pascal statement with the experience spectrum. Where s. exp. = simple expression, c. exp. = complex expression; I.O.E. = Independent Of Experience.

Statement types	Exp. types	Relationship with the experience level	Experience level			
			First year	Second year	Fourth year	Average I.O.E.
IF statement	Factor	Increasing trend	7.62	15.15	20.82	14.53
	Term	No trend	0.19	0.15	0.24	0.19
	s. exp.	Decreasing trend	1.12	1.08	0.80	1.0
	exp.	Decreasing trend	70.45	63.06	62.91	65.47
	c. exp.	Decreasing trend	20.63	20.56	15.24	18.81
CASE statement	Factor	No trend	99.1	99.29	95.0	97.79
	Term	Increasing trend	0.99	0.71	5.0	2.23
	s. exp.	No variation	0.0	0.0	0.0	0.0
	exp.	No variation	0.0	0.0	0.0	0.0
	c. exp.	No variation	0.0	0.0	0.0	0.0
WHILE statement	Factor	No trend	14.1	20.19	16.24	16.84
	Term	No trend	2.56	2.88	0.0	1.81
	s. exp.	Decreasing trend	55.12	6.73	2.62	21.49
	exp.	Increasing trend	10.25	32.69	43.98	28.97
	c. exp.	No trend	17.94	37.5	37.17	30.87
REPEAT statement	Factor	No trend	20.23	11.94	24.56	18.91
	Term	Increasing trend	1.15	1.49	1.75	1.46
	s. exp.	No variation	0.0	0.0	0.0	0.0
	exp.	No trend	35.26	68.65	17.91	40.60
	c. exp.	No trend	43.35	17.91	57.89	39.71
FOR statement	Factor	Decreasing trend	100.0	100.0	93.1	97.7
	Term	Increasing trend	0.0	0.0	6.9	2.3
	s. exp.	No variation	0.0	0.0	0.0	0.0
	exp.	No variation	0.0	0.0	0.0	0.0
	c. exp.	No variation	0.0	0.0	0.0	0.0
ASSIGNMENT statement	Factor	No trend	79.5	84.23	78.22	80.65
	Term	No trend	17.62	11.82	18.36	15.93
	s. exp.	Increasing trend	1.49	1.8	1.81	1.7
	exp.	No trend	0.19	0.22	0.0	0.13
	c. exp.	No trend	1.18	1.93	1.61	1.57

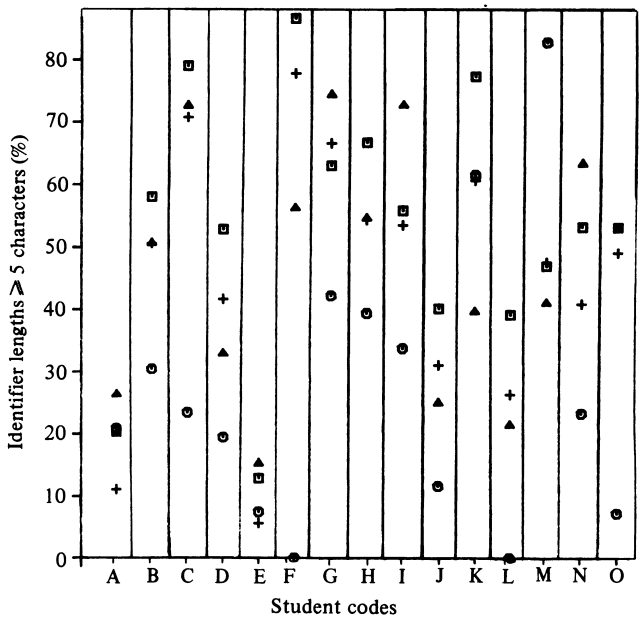


Fig. 22. Student program evaluation vs. long identifier length percentage. ○, First; ▲, Second; +, Third; □, Fourth course work program.

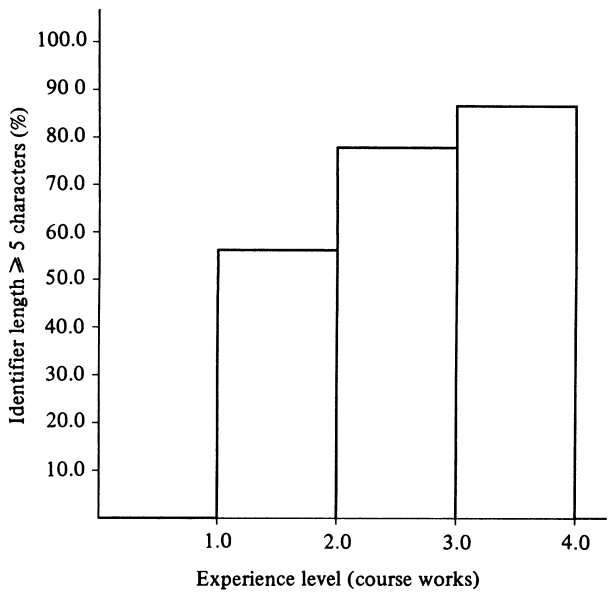


Fig. 23. An example of 'good' student performance. Good improvement for student no. 6.

sample. This suggests that as experience increases there is a tendency to use 'simpler' expressions.

The conventional measure appears to us to be over-simplified. We feel that any analysis of the complexity of expressions should be made in relation to their component expressions and in relation to the statement type that utilises them. However, due to the recursive nature of the expression syntax, no direct conclusion can be extracted from expression components. However, by analysing complete expressions within a statement context, and grouping like expressions into classes, we may observe complexity in a greater detail than the conventional approach offers. We therefore split Pascal expressions systematically into the following classes:

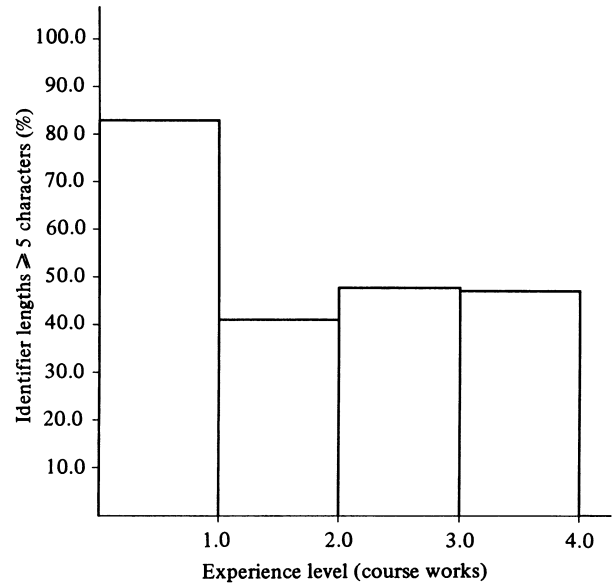


Fig. 24. An example of 'poor' student performance. Poor improvement for student no. 13.

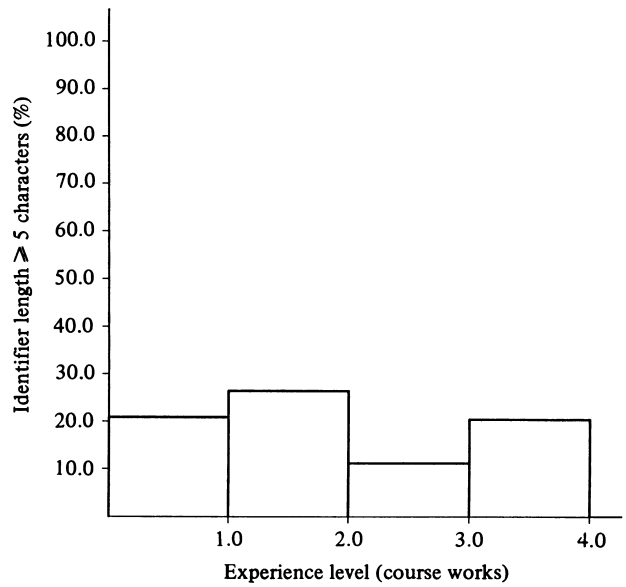


Fig. 25. An example of 'oscillating' student performance. No improvement for student no. 1.

Table 4: The average identifier length percentage for each course-work

Course-work number	Percentage of identifiers exceeding five characters in length
1	26.92
2	45.89
3	46.67
4	53.76

(1) < factor > class: where < factor > :: = < identifier > | < integer constant > | < real constant > | < string constant > | NOT < factor >.

(2) < simple > class: where < simple > :: = < factor > ADDOP < factor >.

Table 5: The experience relationships' prediction of third-year student performance

The measure (%)	Actual value	Regression value	Regression determination factor	Absolute error percentage	A	B
(a) Based on linear regression ($Y = A + BX$)						
ID-length	89.6	93.4	0.95	4.2	79.9	4.4
Average module size	27.6	31.3	0.79	11	40.2	−2.9
User Proc. and func. calls	5.41	7.1	0.96	24	−1.7	2.9
(b) Based on exponential regression ($Y = A^* \exp(B^*X)$)						
ID-length	89.6	93.2	0.95	3.8	80.5	0.05
Average module size	27.6	31.1	0.82	10	40.6	−0.09
User proc. and func. calls	5.41	5.3	0.76	0.5	0.38	0.88
(c) Based on logarithmic regression ($Y = B^* \ln(X)$)						
ID-length	89.6	94.1	0.84	5.0	83.9	9.2
Average module size	27.6	30.5	0.92	9.2	38.2	−7
User proc. and func. calls	5.41	7.9	1.0	31	0.54	6.6
(d) Based on power regression ($Y = A\ (X^{**}B)$)						
ID-length	89.6	88.6	1.0	1.1	85.6	0.03
Average module size	27.6	30.4	0.94	9.0	38.2	−0.2
User proc. and func. calls	5.41	7.0	0.90	23	0.69	2.1

(3) <expression> class: where <expression> :: = <factor> RELOP <factor>.

(4) <term> class: where <term> :: = <factor> MULTOP <factor>

(5) <complex> class.

Fig. 20 illustrates, independent of the utilising statement, how expression class percentage varies with experience level. The resulting percentages are quite close to each other and therefore no useful relation can be derived between expression complexity and experience.

To delve even deeper, another method for determining expression complexity was utilised: that of counting the individual expression tokens. Both strategies were found to exhibit similar behaviour as illustrated in Fig. 21.

Using the first counting strategy, and taking statement context into account, we obtain Table 3. From this we may conclude that the utilisation of expression classes can be related, in some cases, to experience when statement context is taken into account.

7. VALIDATION CASE STUDIES

In order to measure the usefulness of our experience-level relationships, two trials were conducted.

(*a*) *Experience level for consecutive course works.* In this trial we test programmer performance in four consecutive course-works within one undergraduate year (Year 1). We should expect that student experience increases as more programming assignments are completed. We selected fifteen first-year students as our sample (encoded A–O for anonymity). Fig. 22 shows the distribution of one of the experience-dependent measures, the long identifier length percentage, and Table 4 summarises this. Overall we find that average identifier length increases with experience. This suggests that we may use this experience measure to evaluate the performance of each individual student, and then use this as a guideline in the overall assessment of a student's ability. Figs 23–25 show three examples of student 'behaviour' taken from the fifteen-student sample.

(*b*) *Predicting the third-year performance.* In this trial we show that the experience-level relationships can

predict third-year student performance. To act as a control we collected a sample of 26 third-year Pascal programs with a total length of 8,229 lines. This is smaller than the average student-stage sample of 10,900 lines, due to the low variety of Pascal programming assignments given to the third-year students. If the experience measure is reliable, this reduction should be insignificant. The prediction strategy is based on taking metric data from the first-, second- and fourth-year samples and performing various regression analyses. The regression models that prove to be the most suitable (selected on minimal absolute error value and the regression reliability determination factor nearest unity) are as follows:

(*a*) Power regression, to predict the identifier length measure and the average module size (see Table 5*d*).

(*b*) Linear regression, to predict the percentage of user-declared procedure/functions (see Table 5*a*).

As can be seen, the experience-level relationships are generally capable of predicting programming performance and thereby illustrate their reliability.

8. CONCLUSION

An empirical analysis has been performed on university undergraduate Pascal programs. Certain complexity indices are found to be affected by programmer experience, and this may afford quite useful measures for any teaching system.

Acknowledgements

Suggestions from Professor M. L. V. Pitteway, Brunel University, and the referee were invaluable to the development of the final shape of this paper. It is a pleasure to acknowledge the assistance of Miss A. Shrimpton in gathering the data samples.

REFERENCES

1. J. L. McTap, The complexity of an individual program. *AFIPS* **49**, 767–771 (1980).
2. A. Fitzsimmons and T. Love, A review and evaluation of software science. *Computing Surveys* **10** (1), 3–18 (1978).
3. S. K. Robinson and I. S. Torsun, An empirical analysis of Fortran programs. *The Computer Journal* **19** (1), 56–62 (1976).
4. A. S. Tanenbaum, Implications of structured programming for machine architecture. *CACM* **21** (3), 237–246 (1978).
5. M. H. Halstead, *Elements of Software Science*. Elsevier North-Holland, New York (1977).
6. G. Lovegrove and M. J. Rees, Some steps towards automatic teaching and marking. *University Computing* **6**, 16–23 (1984).
7. K. J. Ottenstein, An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin* **8** (4), 30–41 (1976).
8. S. K. Robinson, The study and application of the static and dynamic evaluation of source programs. *Ph.D. Thesis*, Brunel University (1976).
9. A. M. Addyman, Short communication. *Computer Bulletin*, series 2, no. 8, p. 31 (1976).
10. K. Jensen and N. Wirth, *Pascal User Manual and Report*, third edition. Springer-Verlag, Heidelberg (1979).
11. P. Naur (ed.), Revised report on the algorithmic language Algol 60. *CACM* **6** (1), 1–17 (1963).
12. R. H. Perrott and P. S. Dhillon, An experiment with Fortran and Pascal. *Software – Practice and Experience* **11**, 491–496 (1981).
13. M. Shimasaki *et al.*, An analysis of Pascal programs in compiler writing. *Software – Practice and Experience* **10**, 149–157 (1980).
14. G. R. Brooks *et al.*, A static analysis of Pascal programs structures. *Software – Practice and Experience* **12**, 959–963 (1982).
15. H. A. Linstone and M. Turoff (eds.), *The DELPHI Method: Techniques and Applications*. Addison-Wesley, New York (1975).
16. K. V. Roberts, The readability of computer programs. *Computer Bulletin* **10**, 17–24 (1967).
17. J. L. Elshoff, Analysis of some commercial PL/1 programs. *IEEE Transactions on software engineering* **SE-2** (2), 113–120 (1976).
18. C. Shimon, Simplicity = Efficiency = Readability. *Sigplan Notices* **16** (9), 83–89 (1979).
19. D. E. Knuth, An empirical study of Fortran programs. *Software – Practice and Experience* **1**, 105–133 (1971).
20. D. Coleman, *A Structured Programming Approach to Data*. The Macmillan Press, London (1978).
21. D. L. Fisher, Global variables versus local variables. *Software – Practice and Experience* **13**, 467–469 (1983).
22. G. J. Myers, Reliable software through composite design. Van Nostrand Reinhold Publishing Company, London (1975).
23. S. K. Robinson and I. S. Torsun, The automatic measurement of the relative merits of student programs. *Sigplan Notices* **12** (4), 80–93 (1977).
24. H. J. Curnow and B. A. Wichmann, A synthetic benchmark. *The Computer Journal* **19**, 43–49 (1976).
25. J. A. W. Faidhi and S. K. Robinson, An empirical approach for detecting program similarity and plagiarism within a university environment. *The Computers and Education Journal* (1986) (in the Press).
26. J. A. W. Faidhi and S. K. Robinson, Pascal program style analysis and its application to a university environment. *Software – Practice and Experience* (submitted).
27. B. A. Wichmann, The efficiency of Pascal. In *Pascal – The Language and its Implementation*, edited D. W. Barron. Wiley, Chichester (1981).
28. G. Pask, *Learning and Teaching Systems*, edited J. Rose. Butterworths, London (1970).
29. F. J. Lukey, Understanding and debugging programs. *International Journal of Man–Machine Studies* **12**, 189–202 (1980)
30. B. Shneiderman and R. Mayer, Syntactic/semantic interactions in programme behaviour: a model and experimental results. *International Journal of Computer and Information Sciences* **9** (3), 219–238 (1979).
31. R. Brooks, Towards a theory of the cognitive processes in computer programming. *International Journal of Man–Machine Studies* **9**, 737–751 (1977)
32. A. Cowling and J. McGregor, HANDIN – a system for helping with the teaching of programming. *Software – Practice and Experience* **15** (6), 611–622 (1985).