Software Metrics and Plagiarism Detection

Geoff Whale
Department of Computer Science, University of New South Wales, Australia

The reliability of plagiarism detection systems, which try to identify similar programs in large populations, is critically dependent on the choice of program representation. Software metrics conventionally used as representations are described, and the limitations of metrics adapted from software complexity measures are outlined. An application-specific metric is proposed, one that represents the structure of a program as a variable-length profile. Its constituent terms, each recording the control structures in a program fragment, are ordered for efficient comparision. The superior performance of the plagiarism detection system based on this profile is reported, and deriving complexity measures from the profile is discussed.

1. INTRODUCTION

Software metrics have been developed for the purpose of measuring software quality and guiding the software development process. An example of the wide-ranging use of metrics is in the detection of plagiarism in student programming assignments.

A metric is the result of applying a transformation function to an object. In the case of software metrics, the object in question is a program (or a program development process), and the transformation is applied by a software analyzer. The range of potentially useful transformations is very large [1], but most are designed to reduce the object to a small set of symbols that represent some important aspect of the object. The aspect highlighted might be program size, data usage, flow of control, or the rate of errors occurring during development. In all cases, requirements for simple comparison of the complexity of software modules favors the use of metrics that consist of a single numeric quantity.

Plagiarism detection seeks to identify similar programs from a large group implementing the same task. Automated methods for selecting similar student submissions are based on comparisons between reduced representations of the programs. The representations are software metrics, though they are likely to be special-purpose relatives of single-valued complexity measures.

Figure 1 illustrates the three major processes that make up a generalized plagiarism detection system. The metrics generated by the software analyzer are matched to produce a list of program pairs with a measure of the differences between their metric representations. The final phase orders the list on metric similarity and filters out those pairs exhibiting significant metric differences. The submissions listed as potential plagiarisms are ultimately retrieved for manual examination and classification.

2. USING SOFTWARE METRICS FOR PLAGIARISM DETECTION

To be useful in similarity detection, a metric must satisfy four important criteria:

1. It should represent program characteristics that are inherently difficult to change in order to disguise a copy.
2. It should be robust, exhibiting minimal changes due to nonsignificant alterations to the source text.
3. It should be relatively easy to compare.
4. It should be applicable to a range of programming languages.

2.1 Fixed-Size Metrics

The first reported attempt at automated similarity detection is that of Ottenstein [2]. At that time (1977), Halstead's software science [3] was a popular source of software metrics. Unlike previous measures, the software science metrics are derived from a common four-element vector of token counters—the total number of operators and operands in the program and the number of distinct operators and operands. Ottenstein surmised that this 4-tuple (variously known as the "Halstead metric" or "Halstead profile") might be a suitable "signature" and that dependent programs should have identical signatures.

Address correspondence to Dr. Geoff Whale, Department of Computer Science, University of New South Wales, P.O. Box 1, Kensington NSW 2033, Australia.

0164-1212/90/$3.50

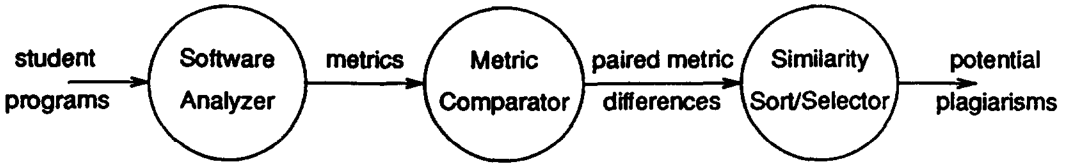

Figure 1. A plagiarism detection system.

Ottenstein found that Fortran programs with identical Halstead profiles were more likely to be related if the constituent counter values were far from the means over all profiles in the population. This implies that the representation provides little discrimination, the primary requirement in this application. Nevertheless, Ottenstein's method is often used as the control method by proponents of other systems.

Much of the effort in the last decade has focused on the search for better metrics of a similar type as the Halstead profile—fixed, multiple counters. Grier [4] adapted the Halstead metric to Pascal programs (extending it to seven parameters by adding code lines, variables, and control statements) and devised a comparison scheme that allows for minor differences between profiles. The first of two methods proposed by Donaldson et al. [5] uses a seven-element vector comprising counts of various statement types, supported by a flexible profile-matching algorithm. The last of the fixed-counter methods [6] resulted from a statistical study of many features that could potentially characterize Fortran programs. Berghel and Sallach ultimately determined that an "alternative" 4-tuple made up of tallies of assignment statements, variables, code lines, and keywords was better at identifying similar programs than the Halstead profile.

The characteristic common to all these approaches is that they discriminate poorly between dependent and unrelated programs. Each author claims to demonstrate improvements over the Halstead metric, but overlooks the fact that only a few of the close profiles are found to be due to plagiarism. Conversely, an indeterminate number of dependencies remain undetected because simple modifications alter the profile substantially. Metrics based on fixed counters inevitably discard so much characteristic material that coincidence becomes the primary reason for resemblances between profiles.

2.2 Variable-Size Metrics

Metrics used for plagiarism detection are not limited to fixed vectors. Donaldson's second method [5] represents a program by a string of symbols, each marking the occurrence of a specific statement type. Similar programs are assumed to produce identical strings, so the system can be defeated by statement reordering. More recently, Jankowitz [7] has described an elaborate system that selects potentially similar Pascal programs for detailed analysis by comparing their static procedure call graphs. The call-graph structure, encoded in a string of binary symbols, is an example of a metric that is too finely tuned for problems of a particular size. Small student programs show insufficient variation in call structure for acceptable discrimination, while the calling sequence in larger programs is too easily disrupted by a minor reorganization of the program.

The deficiencies of the two classes of approach described can be overcome by permitting the representational metric to take on two dimensions—a variable-length sequence of terms each derived from a localized program segment. A scheme of this type has been proposed by Robinson and Soffa [8]. Their system applies code optimization techniques to identify the basic blocks in Fortran programs. The program representation is a vector comprising the number of statements in each basic block. Since the number of basic blocks increases with the size of the program (and is accordingly itself a size metric), this representation is a variable-length profile.

Basic-block vectors are a considerable improvement over fixed-length metrics, as they are sensitive to the sequence of control structures that compose a program. However, they record only extended statement counts, without regard to statement types or to nesting relationships. They are also difficult to match. Unless the vectors are identical, they must be compared using string-matching algorithms that are inherently expensive: since every pair of profiles in a population may need to be compared, such complexity is unacceptable.

2.3 Structure Profiles

One way to permit efficient comparison of variable-length sequences is to order the constituent terms according to some well-defined criterion. The strategy adopted by the author's "Plague" system [9, 10] is to construct a profile from terms derived from the sequence and nesting of control structures occurring a fragment of the program. A profile pair, ordered on attributes of these terms, is able to be correlated on a single pass. The structure profile metric is described in Section 3.

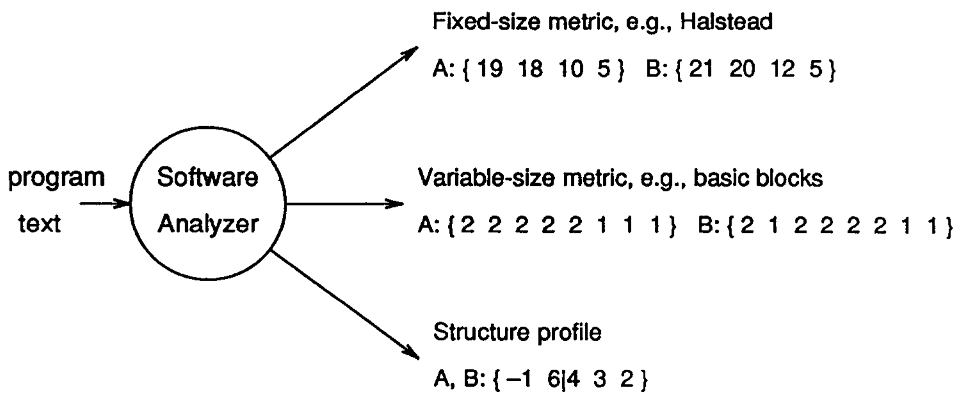
Fixed-size metric, e.g., Halstead
A: { 19 18 10 5 } B: { 21 20 12 5 }
program text
Software Analyzer
Variable-size metric, e.g., basic blocks
A: { 2 2 2 2 2 1 1 1 } B: { 2 1 2 2 2 2 1 1 }
Structure profile
A, B: { −1 6|4 3 2 }

Figure 2. Metrics used in similarity detection.

2.4 Data-Usage Metrics

Apart from the operand counts of software science, metrics suggested for this application have been based on representations of control structure. Although metrics based on data usage have not as yet been proposed for similarity detection, they do warrant brief discussion.

First, data structure representations, derived from program declarations, are unlikely to be satisfactory because of the limited choice available to novice programmers.* The pattern of data usage, however, is more promising. A metric could be constructed from the spans associated with live variables [1], that is, the number of statements between each variable reference. This constitutes a variable-size metric analogous to a basic-block vector. Like basic-block counts, comparisons would be troublesome, and the robustness of the metric is questionable without major modifications to cope with aliasing of variables, redundant operations, and reordering of independent statements.

2.5 Summary

Figure 2 symbolizes the generation of each class of metric from program text. Examples of each metric are derived from the functionally equivalent program fragments of Figure 3. Table 1 summarizes the putative suitability of the three classes of proposed metrics for plagiarism detection according to the properties listed at the start of this section.

3. DERIVATION AND MATCHING OF STRUCTURE PROFILES

3.1 Structure Lists

The first step in obtaining a useful profile of a program's structure is to construct a *structure list,* whose elements are given in Table 2. Resembling a regular expression, the structure list is built during a depth-first traversal of the call graph of the program. The order of procedure calls within the body of each procedure determines the traversal order. Declarations (where separate from executable statements) are not represented. In the structure list notation, a single token represents all simple (nonstructured) statements. Distinctive delimiters are used for iteration and selection, and each can contain multiple components (more common with selection in most programming languages).

The first reference to a procedure is expanded into its definition; subsequent calls are indicated by a single token and a number referring to the position of the definition in the list. Recursive calls and branching statements possess an attribute in the form of the number of static procedure levels in the call graph between source and destination. Figure 4 illustrates structure lists derived from the program fragments of Figure 3.

3.2 Structure Profiles

The structure list is adequate as a representation of program structure, but it is scarcely easier for comparison than the program text itself. While any number of string-matching algorithms could be applied to a pair of structure lists, results would inevitably be poor. This is due first to the small alphabet used and second to the linear nature of the list, where an ordering must be imposed in cases where none really exists, as between the components of a selection statement. Realistic comparisons are possible, however, when small parts of the list are extracted and matched with weak context sensitivity. An ensemble of easily matched terms forms a *structure profile* of the program.

While extracting the profile it is desirable to eliminate nonessential detail. The translation process discards redundant simple statements and neutralizes the nesting of structured statements of the same type. The latter transformation is necessary because selection is often

* Their choice is limited both by their knowledge and by the constraints of the precise problem specifications they are typically given.

Program A:

```
If n > 0 then begin
    r := 1;
    while n <> 0 do begin
        n := n – 1;
        r := r * a
    end
end else if n = 0 then
    if a <> 0 then
        r := 1
    else
        writeln('undefined')
else
    writeln('unimplemented')
```

Program B:

```
If (a = 0) and (n = 0) then
    writeln('undefined')
else if n > 0 then begin
    r := 1;
    while n <> 0 do begin
        r := r * a;
        n := n – 1
    end
end else if n < 0 then
    writeln('unimplemented')
else
    r := 1
```

Figure 3. Equivalent implementations of an exponentiation algorithms

easily modified to disguise a copy (for example, using **if** statements to replace an equivalent **case** in Pascal). Figure 4 shows the effect of these simplifications. Another implicit transformation is the replacement of each procedure call by the structure list of the called procedure. Procedures thus form convenient groupings of statement sequences, rather than modules with solid boundaries.

A structure profile is an ordered sequence of terms derived from a structure list. Each term is either a *statement term*, corresponding to a structured statement with two or more components, or a *component term*, derived from a single component of some structure statement (the outermost structure list is deemed to be a component of a selection statement, as if the program were conditionally executed). Each term possesses three attributes:

Magnitude is a natural number, recording for component terms the exact sequence of nonsimple statements making up the component.† Complete encoding is feasible because of the small number of possible nonsimple statements. As shown in the Appendix, an n-statement sequence has magnitude $O(4^n)$. The magnitude of a statement term is the sum of the magnitudes of its components.

Environment class is either "selection" or "iteration," depending on the type of the enclosing statement for component terms or the type of the statement itself otherwise. It provides a small amount of nesting context, ensuring that a loop body is never matched against a similar selection component.

Arity is simply the number of components associated with a term: one for component terms; two or more for statement terms (a structured statement with one component is fully described by its component term). The purpose of this attribute is to distinguish the two types of terms and to help reduce the possibility of matching unrelated structured statements because of coincidental magnitude.

In the graphic representation of profile terms a leading minus denotes iteration and the arity value for statement terms is suffixed with "|." The profile is ordered by environment class, magnitude, and arity. Ordering is necessary to ensure efficient comparison and is justifiable since the characteristic ordering, that of statement sequences within component boundaries, is encoded in the terms themselves. Finally, repeated terms are recorded once, the repetition factor being prefixed by "*."

Thus the program fragments of Figure 3 generate a seven-term profile, three terms of which correspond to

Table 1. Suitability of Metrics for Plagiarism Detection

	Type of metric		
Property	Fixed size	Variable size	Structure profile
1. Significance of representation	•	•••	••••
2. Robustness	•	••	••••
3. Ease of comparison	••••	•	••
4. Language generality	•••	••	•••

•, low; ••••, high.

† 32-bit machine representation of integers limits the fully encoded sequence to 16 elements, after which only partial encoding occurs.

Table 2. Structure List Notation

Feature	Notation	Pascal examples	Prolog examples
Simple statement	S	assignment Statement	built-in term
Selection	[···|···]	**if, case**	";", multiple clauses
Iteration	{···|···}	**while, repeat, for**	*(not used)*
Procedure call, initial	(···)	procedure call	goal
Procedure call, other	P*n*	*n* refers to initial call	
Recursive procedure call	R*d*	*d* = static recursive depth	
Branch (Pascal **goto**)	G*d*	*d* = static branch depth	*(not used)*

simple statements embedded in selection and are normally deleted. The remaining four-term profile in Figure 4 is representative of all four-way **if** statements with a loop in one branch, a simple loop body, and simple statement sequences elsewhere.

3.3 Profile Matching

The ordering imposed on a structure profile permits matching in linear time. Matching is based on computing the intersection of two profiles and their (symmetric) difference, that is, collecting terms common to both profiles into one multiset, P, and unpaired terms into another, U. A similarity measure is obtained by applying a composite *proximity function* to the terms of the divided multisets. A proximity function consists of a set of weights and simple arithmetic functions that operate on the magnitude of the terms to which they are applied. Separate weights and functions are specified for iteration and selection terms, since the former structure is very much harder to disrupt. Elements of P contribute positive proximity, while those of U contribute negative proximity.

Suitable proximity functions have been selected experimentally from a large palette on the basis of their ability to nominate a high proportion of verified dependent programs. In practice, several proximity functions are applied together, each tuned to different aspects of profile similarity.[‡] Not surprisingly, ideal proximity functions vary among implementation languages, providing the necessary flexibility to cope with a range of programming notations.

4. USING STRUCTURE PROFILES

Kearney et al. [11] stress the importance of designing software metrics to suit the application, rather than hoping that an arbitrarily selected measurement technique will produce meaningful results. The design of the structure profile is based on the premise that the sequence and nesting of control structures is characteristic of dependent programs. The utility of the representation is gauged by the success of the system of which it forms the heart.

4.1 Plague

The Plague system for detecting plagiarism [9, 10] is shown in Figure 5. It relies on an associated system to collect and log student submissions [12]. The instructor initiates a run by specifying parameters for the assignment, such as the programming language used (languages accommodated include Pascal, Modula-2, and Prolog). After the software analyzer has determined structure lists for all submissions, structure profiles are derived and compared. The profiles are used to select, for every submission, a small set of structurally similar neighbors for more detailed analysis. This analysis consists of comparing tokenized versions of the nominated programs using general string-matching techniques. The result is a list of matches ranked by the final similarity measurement computed from the length of the matched portion of the token sequences. The matched pairs are grouped, and the corresponding submissions retrieved, formatted, and presented to the instructor for scrutiny.

The two-stage approach—using structure profiles for the initial filtering and detailed matching for similarity measurement—is very effective. Experimental evidence for its success is shown in Table 3. Several plagiarism detection methods were applied to a population of 245 Pascal programs of about 200 lines each. The table lists the number of related pairs found in the first 10 and first 30 nominations by each method (nominations are in order of apparent similarity as judged by the system). Twenty-four pairs of submissions ultimately were found to be related (some plagiarisms, some cooperative efforts); all but two were nominated by Plague. Other structure-based techniques detected up to a third of the possible matches, while the fixed-metric methods per-

[‡] For example, proximity functions that give low weight to terms in P are highly sensitive to metric differences, while those giving low weight to U respond well to profile subsets.

Structure Lists derived from Figure 3

A: [S{SS}|[[S|S]|S]] B: [S|[S{SS}|[S|S]]]

Structure Lists, reduced

A: [{S}|S|S|S] B: [S|{S}|S|S]

Structure profile from reduced structure lists:	-1	6|4	3	2	1*3
Profile after removal of trivial terms:	-1	6|4	3	2	

Interpretation:	-1	Simple statements embedded in **while** statement.
	6|4	4-way **if**, magnitude 3 + 1 + 1 + 1 = 6
	3	**while** statement embedded in selection.
	2	Outermost structure list (single **if**).

Figure 4. Sample derivation of a structure profile.

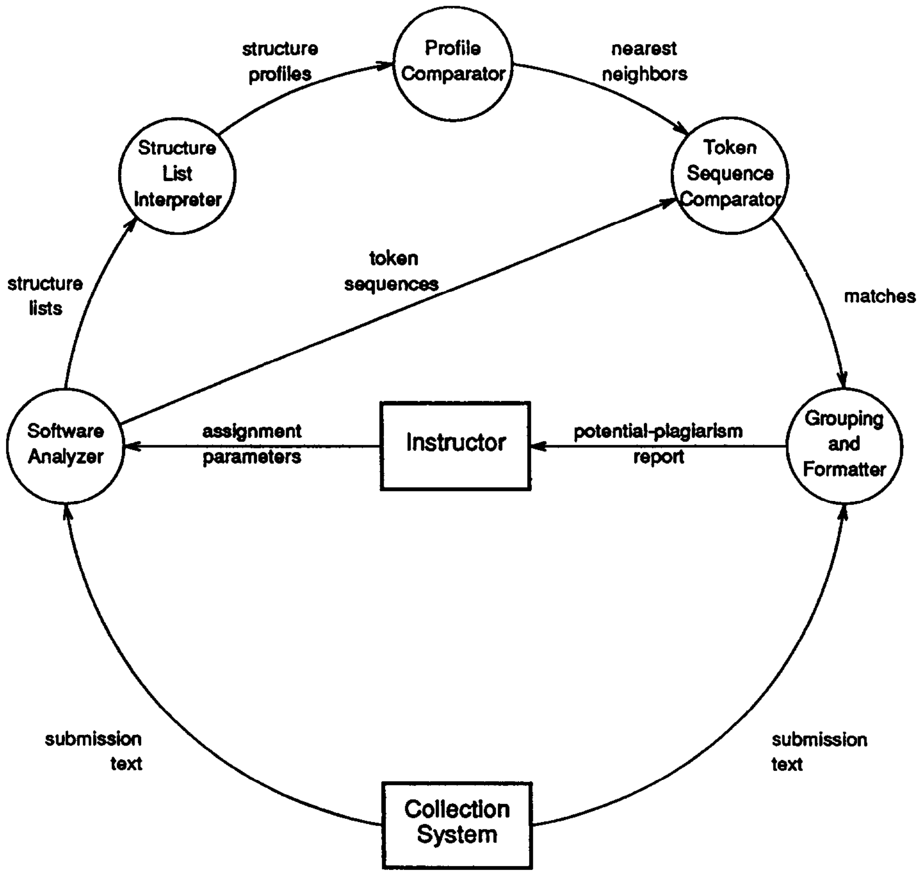

Figure 5. Data flow in Plague (cf. Figure 1).

Table 3. Comparison of Plagiarism Detection Systems

References	Method	Detections in first	
		10 nom.	30 nom.
Ottenstein [2], Halstead [3], Grier [4]	Halstead	1	2
Berghel and Sallach [6]	Alternative	1	1
Donaldson et al [5]	Counters	2	2
Robinson and Soffa [8]	Basic blocks	6	7
Donaldson et al. [5]	Strings	6	8
Whale [9, 10]	Plague	9	22

formed very poorly. A paradigm for comparing quantitatively the performance of similarity-detection systems is described in Ref. 13.

4.2 Derived Complexity Measures

Along with reliable methods for detecting and discouraging plagiarism, there is a desire among some instructors for increased automation of the process of grading student assignments. Because the Plague analyzer is able to generate several metrics for comparison purposes, it is possible to investigate a putative relationship between complexity measures and subjective grades awarded by markers.

The number of terms in a structure profile increases with the total number of control structures. On the other hand, the magnitude of a term increases (exponentially) with the number of nonsimple control structures in a single sequence. The first is a possible size metric, while the maximum magnitude serves as an indicator of localized complexity. These metrics along with the software science volume and difficulty measures and size metrics derived from other representations used for plagiarism detection, were selected to look for a link between complexity and marking.

The results of applying a total of seven measures to a group of 296 student Pascal programs averaging 129 lines were uniformly negative. In agreement with a similar study [14], the various complexity measures correlated well among themselves, but poorly with the marks awarded for good programming practice. No correlation better than 0.12 was observed between style and complexity measurements. These results, while disappointing, bear out Kearney et al.'s claim [11] that meaningful metrics will only appear when a competent model of the process to be measured is developed.

5. CONCLUSIONS

Program representation is the keystone of any system purporting to detect similar programs. Traditional representations for plagiarism detection based on existing software metrics have been quite inadequate, not least because of their adaptation from the fundamentally different task of complexity measurement.

Variable-length measures are capable of recording a variety of program characteristics, and important among these are control structure relationships. The structure profile has formed the basis for a very successful system to monitor and help prevent plagiarism in student assignments. The success of the metric is due primarily to its retention of detail relevant to the application at hand and its exclusion of all incidental material.

ACKNOWLEDGMENTS

I am grateful to Dr. David Carrington for his encouragement and guidance in completing this work.

REFERENCES

1. S. D. Conte, H. E. Dunsmore, V. Y. Shen, and W. M. Zage, A Software Metrics Survey, MCC Technical Report STP-284-86, August 1986.
2. K. J. Ottenstein, An Algorithmic Approach to the Detection and Prevention of Plagiarism, *ACM SIGCSE Bulletin* 8(4), 30–41 (1977).
3. M. H. Halstead, *Elements of Software Science,* North-Holland, New York, 1977.
4. S. Grier, A Tool that Detects Plagiarism in Pascal Programs. *ACM SIGCSE Bulletin* 13(1), 15-20 (1981).
5. J. L. Donaldson et al., A Plagiarism Detection System. *ACM SIGCSE Bulletin* 13(1), 21-25 (1981).
6. H. L. Berghel and D. L. Sallach, Measurements of Program Similarity in Identical Task Environments, *ACM SIGPLAN Notices* 19(8), 65-76 (1984).
7. H. T. Jankowitz, Detecting Plagiarism in Student Pascal Programs, *Computer Journal* 31(1), 1-8 (1988).
8. S. S. Robinson and M. L. Soffa, An Instructional Aid for Student Programs, *ACM SIGCSE Bulletin* 12(1), 118-129 (1980).
9. G. Whale, Detection of Plagiarism in Student Programs, *Proc. 9th Australian Computer Science Conf.* pp 231-241, 1986.
10. G. Whale, Plague: Plagiarism Detection Using Program Structure, Dept. of Computer Science Technical Report 8805, University of NSW, Kensington, Austalia, 1988.
11. J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, Software Complexity Measurement, *Commun. ACM* 29(11), 1044-1050 (1986).
12. D. Carrington, K. Robinson, and G. Whale, Give: A System for Collecting and Testing Student Assignments, *Australian Computer Science Commun.* 6(1), 21-1-21-10 (1984).
13. G. Whale, Identification of Program Similarity in Large Populations, Dept. of Computer Science Technical Report 8618, University of NSW, Kensington, Australia, 1987.
14. D. B. Johnston and A. M. Lister, An Experiment in Software Science, *Lecture Notes in Computer Science* 79, 195-215 (1980).

APPENDIX: STATEMENT SEQUENCE ENCODING

Let L be a sequence of atoms, each of which is either a trivial atom T, or one of k significant atoms S_i, $1 \leq i \leq k$. $\in$ represents the empty sequence and "$\cdot$" denotes concatenation.

The encoding $\xi(L)$ of L is a natural number defined by

$$\xi(\in) = 0$$
$$\xi(T) = 1$$
$$\xi(S_i) = i + 1$$
$$\xi(L \cdot T) = \xi(L),\ L \neq \in$$
$$\xi(L \cdot S_i) = k(\xi(L) - 1) + i + 1,\ L \neq \in$$

To use this encoding method for statement sequences, let T = *simple statement,* and let the $k = 4$ nonsimple statement types be assigned in the order S_1 = *selection* ([$\cdots$]), S_2 = *iteration* ({$\cdots$}), S_3 = *recursive call* (R), and S_4 = *branch* (G). The following table lists the encodings of all one- and two-statement sequences.

S_i	$\xi(S_i)$	$\xi(S_i \cdot S_1)$	$\xi(S_i \cdot S_2)$	$\xi(S_i \cdot S_3)$	$\xi(S_i \cdot S_4)$
S_1 = *selection*	2	6	7	8	9
S_2 = *iteration*	3	10	11	12	13
S_3 = *recursive call*	4	14	15	16	17
S_4 = *branch*	5	18	19	20	21