Detecting Plagiarism in Student Pascal Programs

H. T. JANKOWITZ
Computer Science Department, Southampton University, Southampton SO9 5NH

Detecting plagiarism in student Pascal programs is normally based on a statistical analysis of particular characteristics of style. We discuss how a ‘template’ can be constructed for each program, enabling us to identify other similar templates; this technique is used to extract similar regions or areas from our set of programs in order to perform the statistical analysis. Several examples are presented to show the power of this technique. We conclude by proposing a means of characterising programming style and suggest its use as a gauge for evaluating the student’s assignment.

Received July 1986, revised September 1986

1. INTRODUCTION
Whenever a group of students undertake a programming assignment there is necessarily a certain amount of collaboration during the stages of designing and writing the programs. Indeed, this is widely regarded as a valuable part of the educational process, and at Southampton we encourage students to pool ideas at an early stage, and to discuss amongst themselves the general design of a program. However, when these assignments are marked for credit, it is necessary to distinguish between an acceptable degree of cooperation and deliberate attempts, with or without collusion, to pass off one student’s work as that of another. Such plagiarism can be difficult to detect, particularly if it is disguised by a systematic renaming of identifiers, the changing of layout, and even the changing of the static or dynamic nesting of procedures. This paper presents a novel method for assessing the degree of similarity between two programs, even when they are superficially dissimilar. It is thus able to detect deliberate attempts, through disguising the similarity between two programs, to mislead the instructor. Although developed in this specific context, the methods proposed may have a wider application, since they offer a means of characterising programming style and attaching a ‘fingerprint’ to a program.

Assessment of the similarity between two programs is usually based on a statistical analysis of particular style characteristics, e.g. use of operators, use of special symbols, frequency of occurrence of references to variables, etc. A number of approaches have been reported, mostly based on Halstead’s Elements of Software Science. However, these previous approaches all share one common feature: they analyse a procedure without regard to the context in which it is referenced. In our work the structure of a program, i.e. the order in which procedures are referenced during static execution, plays a fundamental role.

2. THE STATIC EXECUTION TREE
2.1. Construction
The static execution tree (S.E.T.) is a representation, displaying the interconnection between the main program body and all the procedures. It is constructed by parsing the source program: a mapping table is constructed showing which procedures are linked together, i.e. showing for each procedure the procedures it calls and the procedures that call it. The table is assembled without paying any regard to the order in which the procedures might be executed. It does, however, depict the full hierarchical relationship between each and every procedure. By scanning through pairs of such trees, from different programs, similar branches (or subtrees) can be identified. This method establishes the foundation upon which our work is based. The method can be further extended by applying analytical techniques to individual trees as a means of characterising programming style.

3. THE PLAGIARISM MODEL
3.1. Brief overview
After a set of programming assignments has been handed in, before it is assessed by the tutor, it is analysed by the plagiarism program, which performs the analysis in two phases as follows.

(1) Each syntactically correct program is parsed, or scanned, gathering and extracting all the necessary information that is needed to construct a Static Execution Tree;

(2) The statistical data having been successfully collated, pairs of data files are then analysed: initially two static execution trees are created, then having conveniently constructed these trees, they are analysed, scanning for identical branches. The procedures attached to these branches are then analysed statistically.

3.2. Novel approach
The statistical analysis phase is the only part of our method that is similar to those previously referenced. Moreover our approach is more rigorous and flexible, resulting in the model being more accurate and correct. At the end of the analysis the results are recorded for manual inspection.

Note. When a program consists of only three or four procedures, there is little point in constructing an S.E.T. since the tree will be too small to yield any worthwhile information. (This will become clear later on.) When this case arises, the procedures are analysed by comparing each procedure in the first program with every other procedure in the second program.
4. EXTRACTING GROUPS OF PROCEDURES

4.1 Building the static execution tree (S.E.T.)

Each program is parsed so that an S.E.T. can be constructed, in the following manner. The main program body becomes the root of the static execution tree; stemming from it, a branch is added for each unique procedure called from within the main body. The same recursive algorithm is applied to each procedure in turn until that procedure makes no further call to another. Each time a procedural call is made, an addition branch is added. An example of an S.E.T. can be seen in Appendix A. For our purpose a sub-branch DoBlock has been chosen:

```
        DoBlock
       /    /
ReadSymbol DoProcedure DoBody
 /         /
DoProcHgn DoBlock ReadSymbol DoStmt
/  \
ReadSymbol DoTypeDecl DoLabel DoGenStmt
```

Figure 1.

4.2 Manipulating and reconstructing the S.E.T.

Having reached this stage, the tree needs to be altered slightly so that we end up with another tree containing no user-dependent procedure name, yet retaining the original source of information. The method is quite simple; each time a different procedure is encountered, a unique number is assigned to it (subsequently called an atomic token). This number replaces the procedure’s name. By using this technique, we have ensured that all user-dependent information is stripped out; what we are left with is a skeleton structure of the program. In doing this, we have also successfully dealt with cases where procedure names may have been cunningly changed.

```
  0
 / \   
1   2
   /   
   4   0
   /   1
   6   5
   /   
   7   8
```

Figure 2.

The main body is treated as merely another procedure having an atomic token of zero.

Traversing the above tree, using the post-order walk, produces the sequential order in which the procedures are called, or the program’s calling procedure sequence, which is 1 1 6 4 0 2 1 7 8 5 3 0.

The above sequence is still not complete; one further modification still needs to be made. The sequence represents a particular grouping of procedures relative to their absolute position in the S.E.T. In order to make more effective use of the tree, we need to modify it so that when it is parsed we end up with a new sequence, in which the procedure’s atomic tokens are hidden. In doing so we must still be able to associate the now-hidden atomic tokens with the revised sequence. We define a simple mapping function, which will use both the new sequence and the sequence containing the procedure’s atomic tokens, as a means of identifying the procedures. This new sequence will be coded representation depicting the static order in which procedures are referenced in any one program. Furthermore, we will be able to identify modules or subprograms from this new sequence, thereby enabling us to identify groups or blocks of procedures. The connection between the revised sequence and the former sequence is done via the mapping function.

The new sequence is constructed by converting the existing S.E.T. into a strictly binary tree. When the binary tree is traversed, the resulting sequence (or terminating binary sequence) is analysed. This process is explained after the following definition.

4.3 The Terminating Binary Sequence

For a general tree (in our case the S.E.T.), the terminating binary sequence (TBS) can be constructed as follows.

1. Transform the tree into a strictly binary tree.
2. Traverse the resulting strictly binary tree using the post-order walk, and when a node is visited put a 1 in the binary sequence if the node is a branch node and a 0 if the node is a terminal node.

As an example, the TBS for the above DoBlock is produced below (for a full implementation see Appendix B and Appendix C). Traversing the tree, using the post-order walk, produces 0 0 0 0 1 1 0 0 1 0 0 0 1 1 0 0 1 1.

Some interesting properties of the TBS are the following:

1. The original strictly binary tree can be reproduced, without losing any information, from the TBS.
2. A tree with n nodes has a TBS with n 0s and n − 1 1s.
3. A group of k 1s corresponds to a node with k sons in the original tree. (In our model this group represents a procedure which has called k − 1 procedures. The kth 1 is the procedure which has called the k − 1 other procedures.)
4. The zero that terminated the sequence of k 1s corresponds to the node which has k sons. This technique enables us to find, in a given tree, all the occurrences of a subtree with a certain structure. This sequence forms the basis of our analysis.

4.4 Locating identical branches

Having successfully constructed two Static Execution Trees, together with their appropriate mapping functions, we then proceed to extract two identical branches, if they exist. A branch has only one root and will have one or more sons. Only branches with three or more levels are chosen, because anything less than this will cause the operation to be inefficient and moreover will defeat the object of the exercise.

1. We start off by parsing the larger of the two strictly trees, using the post-order walk; this resulting TBS will
henceforth be referred to as the master TBS. Note: this TBS is nothing more than an alternative representation of the strictly binary tree.

(2) The other three is now parsed, extracting its full TBS.

(3) The two terminating binary sequences are now compared, using the Knuth–Pratt string-searching algorithm. The algorithm searches for an occurrence of the smaller string (or TBS) in the master TBS. When a match is found, the starting point in the master TBS is returned; with this information, the corresponding branch in the larger tree can be identified. In addition, the branch corresponding to the smaller TBS can easily be identified and extracted.

(4) When two identical branches have been identified, by using the mapping functions, the respective associated procedures corresponding to 1s in the binary tree are extracted. These procedures are then compared for equality before continuing this process.

(5) If, on the other hand, no pattern was matched, a slightly smaller TBS is extracted from the smaller tree. Each time this phase is entered, a new TBS is constructed by first moving down one branch and then traversing the right-hand branch. If the branch is too small, we backtrack, traversing the left-hand branches.

(6) Steps 2 to 5 are repeated until the whole smaller tree has been searched.

In the above example, the TBS for the branch DoBody is 000011011. This corresponds to the procedures 17853 (ReadSymbol, DoLabel, DoGenStmt, DoStmt, and DoBody). Each 'l' in the sequence represents a procedure, and since the TBS was created from the S.E.T., tracing and identifying the procedures in question is straightforward. The process of matching the 1s in TBS with their corresponding procedures is done using the mapping function. This means it is possible to identify the procedures using their atomic tokens for comparison in the next phase.

4.5 Efficiency of algorithm

The above technique is far more productive and efficient than using a systematic procedure-by-procedure-type comparison. First, the design and structure of the programs plays a fundamental role in the plagiarism detection process. Secondly, the overall number of comparisons is considerably reduced, thus eliminating possible redundant procedure matches. In most of the existing implementations, two statistically similar procedures are often picked out, despite their inherent characteristics being different. This is because the statistical analysis phase in these implementations is not rigorous enough.

An example

A plagiarist wanting to crib the structure of the example in Appendix A would have to alter the sequence in which the procedures were called in order to change the TBS. This might be accomplished by splitting on procedure into two or more parts (or vice versa). In order to disguise the TBS sufficiently, a number of procedures would have to be expanded or joined. Assume an additional procedural cell was made to the existing procedure DoProcHgn (in addition to the few other minor changes, which are going to be used later).

procedure DoProcHgn
var
   symbol: symbols;
begin
   while symbol <> endsy do begin
      ReadSymbol (2);
      for symbol := programsy to typesy do
         process (succ (symbol));
      if symbolname = identifier then begin
         ReadSymbol (1);
         fdeclared := true;
         end;
      DoTypeDec1;
      DoVarDec1;
      end;
   end;
   The reader can verify that the new TBS will be 0000011100000110111. By comparing the two sequences, it can be observed that the latter two-thirds of the terminating binary sequences remain unchanged. Note: although the front parts of the TBS have the same sequence, when the attached procedures are analysed there is little or no similarity between them.

5. ANALYSING THE EXTRACTED PROCEDURES

Once two identical branches have been identified, and their corresponding procedures extracted, their respective bodies are analysed. During the analysis, the statistics gathered in part (a) (see page 2), are compared. This section is divided up into two separate analyses. The first inspects the 'global' characteristics of the procedures. In the second, a more detailed analysis takes place, by paying more attention to the internal structure of the procedure and the composition of the statements therein.

The second part of the analysis is only performed provided the analysis in the first yielded some cohesion.

5.1. General characteristics

The statistics gathered in the first part are:

\[ y_1 \] code lines
\[ y_2 \] variables used (excluding procedure and function calls with parameters)
\[ y_3 \] reserved words (excluding all BEGINs and ENDS)
\[ y_4 \] assignment statements
\[ y_5 \] If statements
\[ y_6 \] REPEAT/WIILE statements
\[ y_7 \] FOR statements
\[ y_8 \] CASE statements
\[ y_9 \] WITH statements
\[ y_{10} \] procedure and function calls

Donaldson et al.2 measured more parameters. The above selected subset has proved to be adequate for our analysis; the task of including other parameters, though, could be implemented without any difficulty. Strictly \( y_1 \) should be written as \( y_{a1} \) and \( y_{b1} \), where \( a \) and \( b \) represent the two procedures under comparison. The number of code lines refers to the actual number of statements in a procedure excluding all I/O statements. The counting of repeat and while statements has been grouped because of their similarity. Items \( y_{a2} - y_{a4} \) are compared with \( y_{b2} - y_{b4} \) in order to get the general flavour of the two
procedures. If potentially similar, this will result in a continued analysis, otherwise the process is abandoned.

For each of the above pairs of values an acceptance region is defined by using standard statistical formulae to determine whether the statistics \( y_{at}, y_{bt} \) lie within the specified range. If they do, the statistics are said to be equivalent. If the lengths \( y_{at}, y_{bt} \) are equivalent (for \( 1 \leq i \leq 10 \)) a favourable counter is incremented by one. If \( y_{at} \) and \( y_{bt} \) are both zero (this case may arise when, say, there are no assignment statements) a null counter is incremented. At the end of this phase a combined evaluation takes place to determine whether or not the process continues into the next phase. The formula used is:

\[
\text{if (favourable-counter/(number-of-lengths-compared - null-counter)) is greater than 50% (or some other accepted value) then continue the process.}
\]

In other words, continue the process if more than 50% of the items are equivalent, with a slight weighting to take into account zero length values. The purpose of this first test is to select only those procedures which, on the surface, are congruous.

Example

Using the original procedure DoProcHgn and the modified one, the results shown in Table 1 were obtained. Applying the ‘process continues’ formula results in

\[
\text{if (5/(10 - 2)) > 0.50 then continue...}
\]

5.2. More specific characteristics

The ensuring analysis constitutes the second phase of the statistical process. This novel approach enables our model to be considerably more accurate. In the models previously reported, the analysis process more or less ended here, although in one case the sequence in which the statements occurred was also taken into account, but this was only done using a very rough algorithm and did not allow for statements to be swapped, added or omitted. Our model includes all of the above, as well as dealing with the order in which statements occur. A more rigorous statistical analysis follows.

The respective procedure bodies are found to be similar, for each IF, REPEAT/WHILE, FOR, CASE and WITH statement, a further analysis takes place, this time inspecting the number of reserved words, variables and other statements referenced. The following measures are recorded.

\[
\begin{align*}
&z_1 &\text{length of statement (simple or compound)} \\
&z_2 &\text{reserved words} \\
&z_3 &\text{variables used in statement} \\
&z_4 &\text{ifs counted} \\
&z_5 &\text{repeat/whiles counted} \\
&z_6 &\text{fors counted} \\
&z_7 &\text{cases counted} \\
&z_8 &\text{wihts counted} \\
&z_9 &\text{reference sequence order}
\end{align*}
\]

For each of the individual statements mentioned, the following tuple can be defined. \( a_i(n) - a \) signifies the procedure in the first S.E.T. and \( i \) refers to the particular statement under observation; \( r \) ranges from 1 to 5, i.e. IFs \( \cdots \) WITHS; \( n \) is the total number of statements of type \( i \) in procedure \( a \).

5.3. Matching statements for comparison

Next, a mapping function is invoked that will pair the statements in procedure \( a \) with those in procedure \( b \) bearing in mind that there might not be a one-to-one mapping of statements from \( a \) to \( b \). There are two tuples, one for each procedure: \( (a_1(n), a_2(n), \ldots, a_r(n)) \) and \( (b_1(m), b_2(m), \ldots, b_r(m)) \).

The first tuple is \( (a_1(2), a_2(1), a_3(0), a_4(0)) \). The second tuple is \( (b_1(1), b_2(1), b_3(0), b_0(0)) \). Associated with each \( a_i(n) \) and \( b_i(m) \) there is a statement sequence order number, relaying information about the order in which the statements appeared in their original context. The reference sequence order is the determining factor when deciding which \( a_i(n) \) cell to compare with the \( b_i(m) \) tuple cell.

Essentially, the task we are facing is: which statements in procedure \( a \) are going to be compared with those in procedure \( b \)? Obviously only similar statements must be considered. Next, there is little sense in comparing a statement in procedure \( a \), occurring as the third statement, with a similar statement in procedure \( b \) if it is, say, the sixth statement. The sequence in which the statements occur must be the prevailing factor, and in our model variations of not more than two units are tolerated. Anything outside this range is regarded as being out of context.

The reference sequence order number is the factor enabling this decision to be made. If two statements, say IF statements, have similar reference sequence order numbers, their individual characteristics are compared using a similar algorithm previously discussed. If the statements differ significantly, before abandoning the analysis for the statements under observation a further check takes place to ensure that there is not another similar statement to compare it with. If not, another state-

Table 1.

<table>
<thead>
<tr>
<th>Counts</th>
<th>Original</th>
<th>Modified</th>
<th>Range</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>12</td>
<td>8</td>
<td>14..10</td>
<td>Fail</td>
</tr>
<tr>
<td>ids</td>
<td>12</td>
<td>9</td>
<td>14..10</td>
<td>Fail</td>
</tr>
<tr>
<td>rws</td>
<td>9</td>
<td>7</td>
<td>7..11</td>
<td>Pass</td>
</tr>
<tr>
<td>Assignments</td>
<td>1</td>
<td>1</td>
<td>2..0</td>
<td>Pass</td>
</tr>
<tr>
<td>IF</td>
<td>2</td>
<td>1</td>
<td>3..1</td>
<td>Pass</td>
</tr>
<tr>
<td>REPEAT/WHILE</td>
<td>1</td>
<td>1</td>
<td>2..0</td>
<td>Pass</td>
</tr>
<tr>
<td>FOR</td>
<td>1</td>
<td>1</td>
<td>2..0</td>
<td>Pass</td>
</tr>
<tr>
<td>CASE</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Null</td>
</tr>
<tr>
<td>WITH</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Null</td>
</tr>
</tbody>
</table>

4 THE COMPUTER JOURNAL, VOL. 31, NO. 1, 1988
Table 2. Using the same two procedures, Table 2 can be derived:

<table>
<thead>
<tr>
<th>Original</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPEAT</td>
<td>WHILE</td>
</tr>
<tr>
<td>IF(1)</td>
<td>8</td>
</tr>
<tr>
<td>IF(2)</td>
<td>3</td>
</tr>
<tr>
<td>FOR</td>
<td>4</td>
</tr>
<tr>
<td>WHILE</td>
<td>2</td>
</tr>
<tr>
<td>FOR</td>
<td>5</td>
</tr>
<tr>
<td>IF</td>
<td></td>
</tr>
</tbody>
</table>

Statement length 10
rws counted 9
ids counted 12
IFS counted 2
REPEATs 0
FORs counted 1
CASEs 0
WITHs 0
Sequence no. 1

ment is then analysed. Each statement of procedure a is mapped as best we can, with a statement in procedure b. After a complete cycle for an \((a_i(n), b_i(m))\) tuple pair, the number of successful matches found is compared with the number of unsuccessful ones, and if there is a general correlation of \(X\%\), then the statements of type \(i\) in procedure \(a\) and \(b\) are said to be cohesive. The procedure bodies are equivalent if the results from each \(a_i(n), b_i(m)\) give a correlation factor above \(Y\%\) (say). Both \(X\%\) and \(Y\%\) are arbitrary values that can regulate the degree of plagiarism.

5.4. Its flexibility

This approach has a certain amount of flexibility providing solutions to problems where: (i) two or more statements might have been swapped; (ii) extra statements have been added (or vice versa); (iii) statements have been expanded or contracted.

6 RECORDING AND INTERPRETING THE RESULTS

After an analysis, if any of the procedures are found to be ‘similar’ the program and procedures’ atomic tokens are printed in the form:

<table>
<thead>
<tr>
<th>Cheating in programs</th>
<th>Assign-13</th>
<th>and</th>
<th>Assign-34</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedures</td>
<td>4</td>
<td>and</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>and</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>and</td>
<td>3</td>
</tr>
</tbody>
</table>

At the end of a run the ‘results’ file would be manually inspected by segregating the cheated assignments. These programs would be further inspected, after which a decision would be made by the assessor. From past experience, one would manually inspect only those assignments in which more than say five procedures were ‘successfully’ matched. This number will obviously depend on the size of the assignment and is a means of regulating the degree of plagiarism detected.

Because of the rather rough set of statistics tabulated in the first part of the statistical analysis, some procedures are matched with which there is insufficient evidence to substantiate a full claim of cheating; this normally occurs when either of the respective procedures is very short, or has a large proportion if I/O statements. Invariably, one would expect some procedures to be similar. In our department, the general rule of thumb is: if more than 50% of the procedures are matched the students are interrogated, otherwise one is hesitant, since in a large class a certain degree of collaboration is expected. Students are encouraged to discuss their work amongst themselves, and this might well be reflected in two programs having a similar static structure; however, as explained above, there are many other considerations to take into account before two procedures are found to be similar.

7. PUTTING THE MODEL INTO PRACTICE

7.1. Philosophy

From the discussion above it should be fairly clear that the static execution tree forms the basis upon which procedures are analysed. It is therefore not surprising that in the course of extracting terminating binary sequences, especially when they get rather small, several matches are going to be found. In most cases no doubt the procedures identified will bear no resemblance; in other cases a few might superficially look the same, and at other times all will be matched. It is latter case we are most interested in.

When the design of an assignment is discussed in a tutorial, or in a lecture, it is not uncommon for the lecturer to give the students a rough top-down design skeleton of the program. The students would then design their programs around this, the design in most cases remaining well within the original framework, which later becomes the S.E.T. If this strategy is used, one would expect many terminating binary sequences to be the same; however, if the mapped procedures were also very similar, one would immediately become suspicious.

7.2 Testing

The above model was first tested on a set of eighty programs roughly between forty and fifty lines in length. Most of the programs did not use procedures as it was an introductory assignment in Pascal. A number of programs were found to be similar: however, no
‘conviction’ was made because of the program’s size and its limited scope for originality. A second set of programs, eighty in all, constituted the next test (their length varied from 300 to 600 lines). The results from this run were accurate and correct. Three cases of plagiarism were detected (in which over 75% of the procedures were the same). In another incident a pair of assignments had over 50% of their procedures matched.

7.3. Results
Initially, students were unaware of the plagiarism programs, which needless to say did cause a certain degree of confusion amongst the students. No doubt thoughts crossed their minds regarding the ‘psychic’ capabilities of their assessor. It is believed that once students became aware of the plagiarism detection programs, the degree of cheating was reduced; this was however difficult to confirm owing to a lack of statistical data. Further testing is planned to take place in the next academic year.

8. CONCLUSIONS
Although the principal aim of the above model is to detect plagiarism in programming assignments, the model has a wider application since it offers a means of characterising programming style and attaching a ‘fingerprint’ to a program.

By developing and applying further analytical models, the design of students’ programs can be monitored and constructively criticised. In particular: (a) commenting on the way local and global variables have been used, providing possible enhancement and improving guidelines; (b) reporting on the usage of parameters during procedure calls and the complexity of their interface.

The most sensitive/volatile procedures can be identified (in relation to their overall position within the S.E.T.), and alternative design strategies might be suggested for particularly volatile ones.

The style aspect, of special interest, centres on designing and structuring programs, thus facilitating an easy to modify, readable and understandable code.

The task of comparing and statistically analysing some of the above-proposed modifications is an additional feature; incorporating these aspects will be relatively straightforward as they fit naturally into our model. As already mentioned, Donaldson et al. measured more style characteristics, a feature soon to be incorporated. Boolean expressions, especially long and complex ones, when analysed during the analysis process, would add to the rigidity of the model. The TBS technique would be applied here, since every expression can be broken down into a tree representation.

Acknowledgements
I am indebted to both my supervisors, David Barron and Gillian Lovegrove, for their support and encouragement in this work. The seed for the ideas was sown by David Barron, to whom I owe the most thanks. Furthermore, I would like to express my appreciation for their help whilst preparing this paper, which covers part of the research leading towards a Ph.D. in Computer Science at Southampton University.

REFERENCES

APPENDIX A
program example1 (input, output);
type
  symbols = (programsy, identifier, colonsy, number, semicolon, endsy, beginsy, procsy, funcsy, varsy, typesy);
var
  symbolname: symbols;
procedure ReadSymbol (numbertoread: integer);
begin
  { gets the next token, storing the type of symbol }
  { in the variable symbolname. }
  { – simple procedure body – }
end;
 procedure DoTypeDecl;
 begin
  { process the type block storing information about } 
  { the various type structures }
  { simple procedure body } 
  end;
 procedure DoBlock;
 procedure DoStmt;
   procedure DoLabel;
   begin
     ReadSymbol (2);
   end;

procedure DoGenStmt;
begin
  \{ process the various statements in this part \}
  \{ i.e. ifs, fors, repeats, etc. \}
end;
procedure DoProcedure;
var
  fdeclared: boolean;
procedure DoProcHgn;
var
  symbol,
  nextsymbol: symbols;
begin \{DoProcHgn\}
  repeat
    ReadSymbol (2);
    if symbol = identifier then
      fdeclared := true;
    for symbol = programs to typesy do
      nextsymbol := succ(Symbol);
      process(nextsymbol);
    end;
    if symbolname = colonsy then begin
      process(colonsy);
      ReadSymbol(1);
    end;
    DoTypeDecl;
    until symbol = endsy;
end \{DoProcHgn\}
begin \{DoProcedure\}
  DoProcHgn;
  if not fdeclared then
    DoBlock;
end \{DoProcedure\};
begin \{DoStmt\}
if symbolname = number then
  DoLabel
else
  DoGenStmt;
if symbolname = semicolon then
  ReadSymbol(1);
end \{DoStmt\};
procedure DoBody;
begin \{DoBody\}
  ReadSymbol(1);
  while symbolname <> endsy do
    DoStmt;
  ReadSymbol(1);
end \{DoBody\};
begin \{DoBlock\}
  while not (symbolname in [beginsy, funcsy, procsy]) do
    ReadSymbol(1);
  while symbolname in [funcsy, procsy] do
    DoProcedure;
  if symbolname = beginsy then
    DoBody;
end \{DoBlock\};
procedure DoProgHgn;
begin \{DoProgHgn\}
  ReadSymbol(1);
  repeat
    ReadSymbol(1);
    until symbolname in [varsy, typesy, procsy, funcsy, beginsy];
  if symbolname = typesy then
    DoTypeDecl;
end \{DoProgHgn\};
begin \{MAIN\}
  while not symbolname in [programsy] do
    ReadSymbol(1);
  if symbolname = programsy then
    DoProgHgn;
    DoBlock
end.

\section*{APPENDIX B. 'ROUGH' TREE REPRESENTATION FOR THE PROGRAM IN APPENDIX A}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{tree.png}
\caption{A rough tree representation for the program in Appendix A.}
\end{figure}
APPENDIX C

Derived strictly binary tree for ‘DoBlock’

Binary tree for ‘DoBlock’

Terminating Binary Sequence:
00001100110001101111