Program Indentation

Paul W. Purdom

TECHNICAL REPORT NO. 54

PROGRAM INDENTATION

PAUL W. PURDOM

REVISED: DECEMBER, 1976

This work was supported by Indiana University and Bell Telephone Laboratories.

Program Indentation*

Abstract: Proper indentation of programs is an important aid for presenting easy to read programs. Previously indentation has been provided manually or with various special purpose programs. Here the basic principles underlying various indentation schemes are presented. The problem of indentation is factored into two parts: providing level numbers to control indentation and separating the text into lines. The first part is dependent on the particular application. It is also straightforward to implement for those languages where the appropriate level numbers are determined from syntactic structure. The second part, while involving some subtle considerations, can be handled by one of several general algorithms. The line-breaking algorithm can be controlled by the level numbers. This paper starts by developing a nomenclature for describing and classifying a large class of indentation algorithms. This class includes close approximations to all the popular methods of indentation and line-breaking. Then the paper discusses the effect of various algorithms in the class. Several which are superior to the others are illustrated. The discussion is useful both for designing a specialized indentation algorithm for a particular language and for designing a general purpose algorithm for inclusion in a compiler building program.

---

*This work was supported by Indiana University and Bell Telephone Laboratories.

1. Introduction

The idea of using indentation to display the structure of programs originated with the development of LISP (1) and ALGOL (2). Previously programming languages had followed the model of assembly language and divided each input record into fields. It is interesting that FORTRAN was compatible with indentation and before 1961 it was a common practice to use indentation to indicate continuation of statements which would not fit on one line (3). Of course, a logical extension of this primitive use of indentation leads, in a language with nested statements, to indentation based on depth of nesting, which in turn provides a display of global structure of the program. Finally by 1965 some authors were indenting FORTRAN to indicate some aspects of global structures (4, 5).

Although proper indentation increases the readability of programs, there are some difficulties in providing it during initial program preparation. Among these is the need to change the indentation of large segments of the program when certain logic errors are corrected. Also, some effort is required to put the indentation blanks on each line. Therefore, there is interest in programs to do indentation.

Programs which provide automatic indentation have been available for some time. Very little, however, has been published on the subject. The conventions used by people are rather informal in nature. Programs which do indentation are often ad hoc in design. They often produce unusual results when fed statements which require several lines for listing. This paper provides a systematic study of indentation, which helps avoid glitches in

indentation algorithms and which leads to general indentation routines suitable for use with compilers for a large number of languages. Such a general indentation algorithm can be a valuable part of a compiler building program.

The problem of formatting a program can be divided into two parts. The first part consists of deciding for each element of the program the level of that element. The level of the first element on a line determines the amount of indentation for that line. The appropriate method for selecting level numbers is closely related to the recursive aspects of the language definition. For algebraic languages there are usually two principle recursions, a statement can contain a series of statements and an expression can contain a number of expressions. Some languages also permit very long identifiers, which will affect the rules for indentation. The details of providing level numbers depend on the particular language one is processing, but it is straightforward for those languages where the indentation information can be obtained from syntactic structure.

Automatic indentation can also be applied to languages such as FORTRAN, where indentation information must be obtained from a direct analysis of program structure (6). Very little has been done, however, to establish the appropriate relation between program structure and indentation. A high quality algorithm for this problem would also be useful for indenting ALGOL programs, where part of the logical structure of programs is contained in the GOTO statements and labels.

The second part of the formatting problem is to break the listing into lines. This feature is needed because the original lines of input may not fit onto a line of output after the leading blanks for identation have been added. In uncrunching application there usually is no indication of where the original line-breaks were. The level numbers needed for indentation can also be quite useful for line-breaking. For example, one common principle of formatting is to avoid line-breaking deep within a structure, which is similar to avoiding breaks at large level numbers. Since such principles are useful for indenting any language, it is possible to design a general purpose line-breaking routine.

The examples of this paper are developed from an ALGOL program by Baumann et. al. (7). The program is shown without any formatting in Figure 1. Under each symbol is the level number. The level numbers in figure 1 have been obtained by setting the level number of the first symbol to 1 and the level number of each successive symbol to that of the previous symbol plus a displacement. The displacement is obtained by summing the following terms: 1) +1 if the previous symbol was the first symbol of a statement, declaration, or program, 2) -1 if the current symbol is end, 3) -n if the previous symbol was the last symbol of n statements (several statements can end on one symbol), and 4) +1 if the previous symbol is end (to balance the -1 part of 2). The semi-colons occur between statements. These simple level numbers will help illustrate most of the important differences between various line-breaking algorithms without introducing unimportant detail. The quality of the line-breaking in the examples would be improved, if a more

complicated set of level numbers, which assigned internal structure to statements, were used.

2. Basic Components for Line-Breaking

There are a number of general principles which can be incorporated into line-breaking. Those indentation programs which occasionally exhibit strange line-breaking behavior usually have neglected one of these principles. Various line-breaking algorithms are obtained by combining a selection of the principles. The general operation of these algorithms is to start at the beginning of the program to be listed, output the longest line consistent with the principles included in the algorithm (or one symbol in case there is no such line or the only such line is empty), and repeat until the entire program is listed. Requiring at least one symbol per line prevents looping while keeping the statement of each principle simple.

This method of presenting line-breaking algorithms permits consideration of a large number of possibilities since the basic components can be easily combined in a large number of ways. Some combinations may, however, imply others, so one should be careful not to include unused pieces when trying to give a simple specification of a particular algorithm.

One can write routines for each principle and a routine to find the longest segment (no longer than the longest possible line) which is consistent with the principles. This approach is useful when experimenting to select a good combination of principles. When one wishes to produce an efficient production program, significant savings can often be obtained by writing a program especially

for the selected combination of principles. An example of this is shown in the appendix, where an explicit program is given for one of the more useful indentation algorithms.

The following principles are rather general. For them to be useful, however, there must be some correlation between the level numbers and the structure of the text to be formatted. Especially, the level numbers for elements within a structure (which is important for indentation) should be no smaller than the first (or alternately last) element. The beginning should be recognizable by the level number increasing just after (or at) the first element. The level number should come back to the original value near the end of the structure. The details of defining level numbers will be discussed later.

The following set of principles have been selected to include all those that one is likely to consider for a level driven line-breaking program. The usefulness of various combinations of the principles will be considered in the next section. Each principle except number 2 has a short example in parenthesis. In the example symbols are represented by one digit level numbers. It is assumed that each line can hold 5 symbols. The end of each 5 symbols is indicated by a letter E. The place where the principle will break the line is indicated by the letter B. Therefore the example for principle 5, (1111B2E33), shows that if the input consists of four symbols with a level of one, a symbol with level 2, and two symbols with level 3 then for a line that can hold 5 symbols, principle 5 will cause the first four symbols to go on a line.

The principles to select from, to build a line-breaking algorithm to be driven by level numbers, are organized by function as follows:

Basic

1. The output line can be no longer than some limit, typically 120 characters (11111BE). This principle should be included in all algorithms. The examples use a line limit of 35 characters, long enough to be realistic but short enough for small examples to illustrate some problems.

2. The line should not be broken in the middle of a symbol. Actually special provisions should be made for very long symbols. This principle along with the appropriate special provisions should be included in each algorithm.

Display of Structure

3. (basic) A line cannot contain a symbol with a level number less than the level number of the first symbol on the line if that symbol is immediately followed by a symbol with a larger level number (21B2). This condition is a weak condition that will ensure that the closing of each multi-line structure will be displayed. It should be included (or be implicit) in each indentation algorithm.

4. (show end) A line cannot contain a symbol with a level number less than that of the first symbol on the line (2B1). This implies principle 3. It ensures that each level on the way out of a multi-line structure will have its own line.

5. (start with front) A line cannot end in a symbol followed by a symbol of higher level number unless the second symbol is followed by a symbol of even higher level number (1111B2E33). If the level numbers are designed to increase just before (instead of just after) the first symbol of a structure, then this principle should be replaced by the principle that a line cannot end in a symbol preceded by a symbol of lower level number unless it is also

followed by a symbol of higher level number (1112B3E3). This principle will move one symbol to the next line when it is necessary to cause a line to start at a structure boundary. This principle, of course, can be overridden by the requirement that each line have at least one symbol.

6. (no front at end) A line cannot end with a symbol which is immediately followed by a symbol of higher level number (111B12E). For level numbers that increase just before the structure beginnings, this principle should be changed to a line cannot end with a symbol immediately preceded by a symbol of lower level number (1111B2E). This principle implies principle 5. It prevents multi-symbol lines from ending on the first symbol of a structure.

7. (show front) A line cannot end with a symbol which has a level number greater than that of the first symbol (1B2). When structures have both ends at the same level, this principle causes each level going into a multi-level structure to have its own line.

8. (level line) A line must contain no symbols with a level number less than that of the first symbol (232B1). It must either end with a symbol that has the same level of the first symbol or it must be followed by a symbol that has the same or lower level as the first symbol. This principle implies principles 4 and 7. It causes each level going into and out of a multi-line structure to have its own line.

9. (no substructure) Each line can contain only two levels (12B3). For level numbers that increase just before the first symbol of a structure, this principle should be modified to permit only one level per line (1B2). This principle prevents a line from containing imbedded structure.

10. (one structure) A line can contain only one sequence of symbols with level numbers greater than that of the start symbol (1231B2). This causes each line to contain only one structure of the outermost level.

Shallow Breaks

11. (end fix) A line cannot end in a sequence of symbols which can be moved to the next line if this action will both decrease the level number of the first symbol for the next line and not cause any symbols to be removed from the end of the next line (symbols may be added to the next line) (2B121, assuming principle 3 is also used). This principle requires consideration of the next line while deciding how to format the current line. The recursive version will be called 11r. It does appear to be suitable for use with a one pass indenter. Principle 11 or 11r gives a way to prevent deep breaks in those cases that can be handled without increasing the number of lines of output.

12. (shallow end) A line which contains a symbol with a level number less than or equal to that of the preceding symbol must end on a symbol of smallest level number among those symbols with a level number less than or equal to that of the preceding symbol (121B32E). This principle helps prevent a line from breaking in a deep structure at the end if the line contains complete structure.

13. (shallow break) A line-break cannot occur in a sequence of symbols short enough to fit onto one indented line if that sequence obeys principle 8 (level line) (11B222E221). This principle causes those structures which will fit on a line to be given a line.

3. Basic Line-Breaking Algorithms

Now various line-breaking algorithms that can be formed from the basic principles will be considered. These algorithms break lines with no information other than the level numbers of the symbols. Later algorithms with additional features will be considered. The discussion will assume level numbers increase on one symbol after the start of a structure, although the situation is quite similar for level numbers that increase at the start of a structure provided the appropriate alternative is taken when carrying out some of the principles.

To ensure proper termination, in some cases it will be necessary to assume the end of the input is followed by a non-printing symbol with level number minus infinity. Also special action may be needed to force the printing of the last line or two. Basically the special symbol after the end should act like it needs to go on a line by itself.

All the algorithms in the class being considered can be implemented as one pass algorithms provided they do not include principle 11r. A buffer long enough to hold three lines of symbols and their associated level numbers will be needed in some cases. One line of storage is needed for the current line. For some algorithms one line of storage is needed for the next line. Finally for some algorithms it is necessary to look one symbol past the second line. These limits assume that special action will be taken for symbols more than one line long.

With line-breaking algorithms there is a trade-off between length of the listing produced and the detail of structure which

is displayed since display of structure uses up space on a listing. A short listing is a help when studying long programs; so is display of structure. In dentation of detail structure helps understand details of a program, but may camouflage some of the information displayed about global structure. Thus there are a variety of preferences for the compactness versus level of detail appropriate for an indentation program. A poor algorithm, of course, can produce a long listing while not displaying structure well. In the following, special attention is given to those algorithms that produce compact listings, display detail well, or do some combination of those well.

All the algorithms considered contain principles 1, 2 and 3 (for some it is not necessary to include 3 explicitly since it is implied by other principles). The basic algorithm includes only these principles, and its effect on the sample program is shown in Figure 2. The listing is quite compact and yet is considerably more readable than the unformatted listing. The structure of multi-line statements is shown, although not in the normal way. This simple algorithm has two defects. First structure is not displayed as explicitly as most people would like. Second it does not avoid breaking lines deep within structures. Since one can obtain an even more compact listing while occasionally choosing better break points, there is no reason to recommend the basic algorithm.

The end fix algorithm uses principles 1, 2, 3 and 11. The effect of this algorithm is shown in Figure 3. There is a lot of similarity between figures 2 and 3. Close scrutiny shows

that the differences result in a slight improvement in the display of structure in Figure 3. Also the listing in figure 3 is one line shorter than the listing in Figure 2. Principle 11 is a useful addition to most indentation algorithms since it will usually improve both the selection of line-breaking points and the compactness of listings. The end fix algorithm is recommended for those that want very compact program listing.

One needs, however, to look to other possibilities if one wants more display of strucuure. An algorithm with principles 1, 2, and 4 displays the end of a multi-line structure. Since it does not avoid deep break points, however, a lot of space is wasted by displaying a lot of detail that should be suppressed. Although the effect would not be too bad with the sample program, it would be much worse if the level numbers had been chosen to reflect the internal structure of statements. This algorithm can be greatly improved by including principle 11. In addition, adding principle 5 helps. There is no reason to recommend this algorithm.

An algorithm with principles 1, 2, 3, and 7 will display the front of multi-line structures. Again adding principle 11 results in a big improvement. There is no reason to recommend this algorithm.

The same level algorithm uses principles 1, 2, 6, and 8. The short same level algorithm adds principle 10. Those that like the same level algorithm probably prefer the short version. Principle 6 prevents the beginning of a structure from being included on the end of a line. Principle 8 forces complete structures onto a line when possible. Principle 10 limits a line to one structure at the outermost level. The effect of the short same level algorithm is shown in Figure 4. The long version would have combined

lines 2 and 3. This algorithm has considerable merit. The beginning and end of each multi-line structure is clearly indicated. Although the author knows of no established style of indentation which is the same as that produced by the short same level algorithm, it does treat begin and end exactly the same way as they are handled in a popular style of indenting ALGOL. The only defect of this algorithm is the long listing that it produces. While many people initially conclude that this algorithm gives strange indentation of long statements, they usually come to like all aspects of it after a little consideration.

Replacing principle 8 with principle 9 can result in an even longer listing with all structure displayed. For the sample program, the results would be the same as Figure 4. On the other hand, if the level numbers had been assigned to reflect the internal structure of statements, this algorithm would have produced a much longer listing and it would have displayed structure in too much detail. (The short level line algorithm with such level numbers on the other hand would have limited itself to minor changes to improve line-breaking within statements.) This algorithm is only useful when the level numbers do not provide too much detail about the structure of the input. This defect limits its usefulness as a general purpose indentation method.

The end fix algorithm produces compact listings by avoiding deep breaks in lines when this does not lengthen the listing. The level line algorithm produces long listings with detail structure by having each line contain a beginning element, some ending elements, or a complete structure. There are several interesting algorithms which produce results of intermediate compactness by always avoiding deep breaks without requiring level lines.

The basic level end algorithm uses principles 1, 2, 3, and 12. Principle 12 prevents end of the line from containing increasing level numbers unless the whole line is made up of increasing level numbers. Adding principle 6 produces the level end algorithm. It prevents lines from ending with the first element of a structure. The effect of the level end algorithm is shown in Figure 5. This algorithm is very popular. Most LISP programs are indented with it or one of its minor variations.

The basic shallow break algorithm uses principles 1, 2, 3, and 13. Principle 13 prevents structures that will fit on one line from being broken. The shallow break algorithm also has principle 6. It happens that Figure 5 also shows the effect of this algorithm. The appendix contains a program for the level end with shallow breaks algorithm, which uses principles 1, 2, 3, 6, 12 and 13. The algorithm also has provisions for forced line-breaks (discussed later). It does a very good job of indenting LISP programs. It will also handle ALGOL correctly if forced line-breaks are used before ends. Furthermore, it can be easily changed into several other interesting algorithms. Principle 13 is a useful addition to most indenting algorithms.

The shallow break with explicit level ends uses principles 1, 2, 3, 4, 6, 12 and 13. Principle 4 causes the end of a multi-line structure to have its own line. The effect of this algorithm is shown in figure 6. The results are quite close to those of a standard way of indenting ALGOL. Adding internal structure to statements would result in a listing just like ALGOL except for the strange placement of some semi-colons. Even this difference would

go away if the level numbers were assigned to decrease after semi-colons instead of at them.

These examples show that one can obtain a number of high quality indenting algorithms using no information other than level numbers. Yet it is also clear that languages can have various structures but which may need to be indented differently. Thus, it is difficult to handle both end and semi-colon correctly in ALGOL with level number algorithms. Since there are some straightforward high quality level number algorithms some people may want to switch. In the next section some additional features are discussed, which will be useful if one wishes to implement his favorite indentation scheme.

4. Additional Features

Permitting the first phase of an indenting program to force line-breaks permits added flexibility. With this feature the first phase can decide to break lines at a place in some structures without deciding to break at the corresponding place in other structures. Without this feature one might have difficulty producing this effect while still having both structures indented correctly. Even in systems with forced line-breaks one normally wants to use one of the preceding line-break algorithms to break long sequences which do not contain a forced line-break. Although with most languages it is easy to select points for forced line-breaks that will usually produce segments less than one line long it is usually difficult to select good points which will always produce short segments. A line-break algorithm for use with forced line-breaks should,

of course, be selected to not produce too many breaks since the feature can be used to add additional breaks but not to prevent breaks. Forced line-breaks give great flexibility but should be used carefully to avoid producing an overly complex system.

The sample program, as originally formatted by Baumann et. al. (7), required a line-break before each statement and before each end. They also used a fairly long line and the level end with shallow breaks algorithm (or a close approximation to it). Several other programs in the same book also use this style. There are, however, several other slightly different styles of indentation used in other parts of the book. The common ways of indenting ALGOL show how that occasional use of forced line-breaks can produce useful indentation methods.

Another feature that is occasionally useful is a multi-component level number. By using only the first component for indentation control while using the entire number (with lexicographic comparison) for line-breaking. This permits indentation to be controlled only by the gross structure while the fine structure can still control line-breaking.

Some authors vary the amount of indentation so that the first symbol on one line will line up with the symbol at that level on the previous line that had a symbol at that level. Thus "begin integer a; real x;" would be formatted with the r in real under the i in integer. This can be handled by adding to the preceding algorithms a table to convert level numbers into indenting amounts. The entries are set going into structures and reset to empty going

out. If the beginning of a line has a higher level than the end of the previous line (resulting in empty beginning in required entry in the conversion table) then the new line is started one column to the right of where the previous line ended. This method of indentation works well with simple programs, but causes the indentation to approach the right margin too quickly with deeply indented structures.

Another approach to line-breaking is to set up preferred break-points according to features in the input. The break-points can have a desirability ordering. The line is broken at the rightmost (or leftmost) break-point of the most desirable type. In its pure form this approach does not work well with languages that have recursive structure. For example, one may prefer to break lines at plus rather than at times. Yet with the input "a+b×(c+d)" it is better to break at the times sign than to break at the second plus. Once this defect is cured one obtains a method that is like the two component level number method.

The problem of how to divide lines into pages is similar to the problem of dividing symbols into lines. The current practice is to either fill each page up or else to go to a new page at the start of each routine in the program being listed. There are a lot of advantages to avoiding deep breaks when going from one page to the next. The previous algorithms can be used to divide listings into pages provided all lengths are measured in lines rather than symbols. For this application one wants an algorithm which produces compact results, such as the end fix or shallow break algorithm.

5. Syntactic Generation of Level Numbers

When one is generating level numbers based on the syntax of the input, basically one wants the level number to increase near the beginning of the structure and to come back to the same level near the end. In between, the level numbers should not go below the starting value and care must be taken when they are equal to the starting value. Thus an if statement that has if, then and else on the same level (with everything else at a higher level) may be intended as though it was three statements. Usually one wants to have the level number increase one symbol after the beginning of a structure so that the tail of the structure will be indented if the structure is broken in the middle. Often the level number should come to the starting value one symbol after the end of the structure. If the structure ends in a special word, such as end, whose function is to indicate the end of the structure, it is more natural to return to the starting level at that symbol. Occasionally one wants to decrease the level number at the beginning of a structure (such as label) to make it stand out. These basic features can be used to provide proper level numbers.

To produce the level numbers one needs routines delayedincrease and decrease to call during parsing. The routines affect two variables, level and levelincrement. When processing a symbol, level is increased by levelincrement and then levelincrement is reset to zero. The value of level is the level number for that symbol. The routines are called between symbols. The routine delayedincrease is usually called before the first symbol of a structure. It increases levelincrement by one. The routine decrease is usually

called at the end of a structure. It increases level by levelincrement minus one and resets levelincrement to zero. The reason decrease interacts with levelincrement is to obtain a reasonable effect on structures with only zero or one symbols. (If one is a perfectionist, one needs to record a maximum and minimum value of the level number for each symbol, and one needs to modify the previous algorithms to use the level number pairs, so that very short structures will be noticed by the indentation method.) The two basic functions delayedincrease and decrease will serve most needs for generating level numbers.

For some uses it is convenient to have an ordinary increase or a decreaseprevious function (if the latter one is provided the line-break algorithm must be careful not to believe a level number until it stops changing). Often it will be useful to have a function to force a line-break. Finally for reasons of efficiency one may wish special functions that have the effect of two calls to the basic level number functions. Thus a function for decrease followed by delayedincrease could be called before a label or end. For use with two component level numbers one needs functions to change the second component and functions to change the first component while resetting the second component. Also in a multi-com component system, if one component deals only with non-recursive parts of the language, it may be useful to have functions to set the component to a fixed value (this is similar to the idea of levels of preferred break points mentioned earlier). One should, however, stick to the basic routines unless one has a need for one of these more complicated actions.

The most natural way to indent a programming language is to call delayedincrease at the start of each production of the grammar for the language and to call decrease at the end. This will never cause trouble for LL(k) grammars (8). It will also work for many LR(k) grammars (9) which are not LL(k). The parser at the start of a phrase must only know how many phrases are being started, not which ones. Indenting on every production, however, will often produce too much indentation, so one may want to either indent only certain productions or else use a multi-component method. The production important for indentation usually starts with key words, which removes all possible problems with the initial call to a level number function. When the production ends with a terminal symbol, that symbol often has the function of indicating the end of the structure, and in that case one may wish to decrease the level number before the symbol instead of after it. Usually there is no trouble having the parsing program call the level number routines and the forced line-break routines at an appropriate time.

6. Level Numbers from Structure

There are many languages which cannot be effectively indented using only syntactic structure. Included are FORTRAN and SNOBOL. For such languages to be indented effectively one must analyze the flow of the program. A good technique for indenting based on program flow would also be useful for indenting languages such as ALGOL

and PASCAL where part of the flow information is contained in GOTO statements.

FORTRAN is the main language which is occasionally indented to show structure and which has most of the flow of control information in GOTO statements. The current indentation practice for FORTRAN is not nearly as well established as that for the more structured languages. The basic goal appears to be to imitate the indentation that would be obtained if the program were converted to ALGOL. Therefore loops are indented and sequences with the format of if then statements have the part that can be skipped over indented. Structures that are as complex as if then else statements have no established FORTRAN formatting style. Furthermore when identifying which sequences look similar to a corresponding ALGOL structure one has a choice of using information relating to the linear ordering of the blocks of the program or of basing the identification on the graphical properties of the flow structure.

The work of Lowry and Medlock (6) contains one suggestion on how to indent FORTRAN programs. Although their paper has a good algorithm for identifying loops their indentation algorithm does not indent loops. The work of Baker (10) also implies a method of indenting FORTRAN since the transformations she applies could be revised to indent the original program (according to the way the transformed program is indented) instead of transforming it into a new language.

The algorithm the author likes best for indenting FORTRAN programs is a further development of the one of Lowry and Medlock (6). First the immediate predominator and the nesting level for each block is computed. Then the initial block is assigned a level

of one. After the level for a block is set, the level of the nodes that it immediately predominates are set. First these predominated nodes are grouped into maximal equivalence classes containing blocks that can reach other blocks by paths that do not go through the immediate predominator. Each equivalence class which can reach no other equivalence class (by paths of the type considered above) is given a level number. If there is one such class it gets the same level number as its immediate predominator. Otherwise these classes get a level number one higher than the immediate predominator. For the remaining classes notice which classes they can reach by paths which do not go through an immediate predominator or through the blocks of any of the blocks belonging to the classes. Each remaining class is assigned a level number one higher than the level number of the highest class it can reach. Each block in a class is assigned the same level number as the class. After all the level numbers are formed in this way, each one is increased by the nesting level of its block. This gives the final indentation level.

This algorithm will indent loops, if statements, and case statement equivalent structures in the normal way. The loops are defined as in Lowry and Medlock (6) so that they have one entry, rather than the way they are defined by Baker (10) (where they include all backwards transfers). The results of this algorithm are shown on a program in the appendix, which has a rather complex flow structure.

An interesting variation to the above indentation algorithm is to first collapse each loop into a single block. The resulting structure is indented with the above algorithm. Then each loop is considered separately and the process repeated for each loop. The

level for blocks in a loop is the sum of the level it was assigned when the loop was considered separately and the level assigned to the loop when the whole loop was considered as a block. For those programs that can be presented as while until loops this method will give a result similar to that provided by indenting a while until program (11). For the program in the appendix both methods give the same results.

7. Practical Considerations

There are three practical problems that should be handled by any formatting system. The first is very long symbols. These can occur in most languages (consider Hollerith strings in FORTRAN). Even when the language definition does not permit long symbols, they may be present in the input due to errors. Many languages do not permit blanks in a symbol and do not insert a character at the end of a line. For these languages it is appropriate to suppress some indentation when it permits a symbol to fit on one line. For symbols that are too long to fit on a line, the indenter should go into a special mode where all of the symbol after the first line is started in the first column (the first column should be blank at all other times) and continues in full lines until the end of the symbol. When meaningless blanks are permitted in symbols, the symbols can be broken into maximum length pieces which can be indented normally (with extra indentation for the continuation segments). For languages which insert a character after each line it may be possible to avoid the very long symbol problem.

The second problem is large level numbers. One way to ensure that there is room to list the program is to have a maximum effective level number. Level numbers above this cause no more indentation. One also does indentation with the level number modulo the maximum. With a two pass algorithm one can first calculate the maximum amount of indentation that will be needed. Then an amount of indentation per level can be calculated. With this approach deeply indented programs get one unit of indentation only after several units of change in the level number. Although deep indentation does not occur frequently, some provision should be made for it.

Finally an indentation algorithm should take appropriate action if the level numbers start to go below the initial level. Although this should never happen with a well designed level system and legal input, the indentation program should do something reasonable with illegal inputs. An appropriate action is to permit the level number to become one less than the initial level, but no lower. This will give a visual indication of the trouble.

8. Conclusions

A moderate number of basic principles have been presented which can be combined to provide algorithms for formatting programs. These are useful both for devising new algorithms and for categorizing existing algorithms. These methods divide the problem into providing level numbers and then breaking lines based on the level numbers. The provision of level numbers depends on the language being formatted. Line-breaking can be handled with general purpose algorithms.

Three algorithms deserve special mention. The level end algorithm is the standard method for indenting LISP. Adding the shallow break condition gives an improved variation. Some people prefer the short version of the algorithm. The algorithm is also quite good for formatting ALGOL in a standard way provided forced line-breaks are used. The short level line algorithm gives a very simple consistent way to display all the structure that won't fit on a line. It is the best algorithm when the length of the listing is not very important. Finally the end fix algorithm is the one to use if one wants to compact listing while maintaining a minimally adequate display of structure. Usually one will want a variation of one of these three algorithms.

Acknowledgements: I wish to thank Dick McCullough who corrected several errors in the program in the appendix.

Appendix

The following FORTRAN, routine, OUTSYM, will indent programs according to the level end with shallow break algorithm. It depends on the parser to read the input, store it in the buffer CHARB, to calculate the level number for each symbol, and to zero the buffer LEVELB. The parser should call OUTSYM after each symbol (although it would be simple to remove this condition). Also the parser must handle symbols that are too long to fit on a line and prevent the level numbers from being less than or equal to zero. Finally the routine assumes the input has no leading blanks, i.e. CHARB (1) ≠ blank.

```
      SUBROUTINE OUTSYM
C THE MAIN ROUTINE FOR FORMATING LISTINGS WITH THE SHALLOW BREAK
C ALGORITHM.
C THE VARIABLES IN THE NEXT COMMON BLOCK ARE PARAMETERS TO THE ROUTINE
C   AND ARE OFTEN SET BY OTHER ROUTINES.
      COMMON /COUTS/ LASTC,LASTS,LEVEL,NEWLIN,CHARB(480),LEVELB(480)
      INTEGER LASTC,LASTS,LEVEL,NEWLIN,CHARB,LEVELB
C LASTC, THE POSITION IN CHARB OF THE LAST CHARACTER OF THE LAST SYMBOL
C   WHICH HAS BEEN PROCESSED BY THE PARSER. THIS SYMBOL IS SET BY THE
C   PARSER AS IT PROCESSES EACH SYMBOL. IT IS RESET BY THIS ROUTINE AS
C  THE BUFFER CHARB IS PARTLY EMPTIED.
C LASTS. THE POSITION IN CHARB OF THE LAST CHARCATER OF THE LAST SYMBOL
C  WHICH HAS BEEN PUT IN THE BUFFER. IT IS SET BY THE SCANNER AND RESET
C  BY THIS ROUTINE WHEN CHARB IS PARTLY EMPTIED.
C LEVEL, THE INDENTATION LEVEL OF THE CURRENT SYMBOL. IT IS SET BY THE
C   PARSER THE FIRST TIME A SYMBOL IS NEEDED, AND BY THIS ROUTINE WHEN
C   IT IS NEEDED AGAIN. THE VALUE ZERO IS USED AS A CODE TO FORCE ALL OF
C   THE CURRENT BUFFER TO BE PRINTED.
C NEWLIN, A VALUE GREATER THAN ZERO INDICATES THAT ALL OF THE CURRENT
C  BUFFER IS TO BE PRINTED, SO THAT ANY REMAINING TEXT THAT HAS NOT YET
C   BEEN PUT INTO THE BUFFER WILL GO ONTO A NEW LINE.
C CHARB(4*LINEL), A BUFFER TO STORE THE CHARACTERS WHICH HAVE BEEN READ
C   BUT NOT PRINTED. IT IS FILLED BY THE PARSER. THE CHARACTERS ARE
C   MOVED BY THIS ROUTINE WHEN PART OF THE BUFFER IS PRINTED. THE LENGTH
C   NEEDS TO BE FOUR TIMES LINEL.
C LEVELB(4*LINEL), A BUFFER TO STORE THE LEVEL OF EACH SYMBOL IN THE
C   POSITION MATCHING THE LAST CHARACTER OF THE SYMBOL IN CHARB. IT IS
C   SET BY THE PARSER. ZERO IS STORED IN THE OTHER POSITIONS, UP TO
C   LASTS. THE LENGTH SHOULD BE THE SAME AS CHARB.
C THE VARIABLES IN THE NEXT COMMON BLOCK ARE LOCAL VARIABLES OR
C   CONSTANTS. THEY ARE PUT INTO COMMON SO THAT THE PROGRAM WILL WORK ON
C   FORTRAN VERSIONS THAT DO NOT SAVE VARIABLES FROM ONE CALL OF A
C   SUBROUTINE TO THE NEXT. NOTE HOWEVER THAT OUTST AND TENTAT MUST BE
C   INITIALIZED BEFORE THE ROUTINE IS CALLED.
      COMMON /COSLOC/ BLANK,COLUMN,INDENT,INDFAC,LEVELL,LINEL,MAXINT,
     1  MAXLEN,OLDCOL,OLDLEV,OUTST,STARTC,STARTL,TENTAT
      INTEGER BLANK,COLUMN,INDENT,INDFAC,LEVELL,LINEL,MAXINT,MAXLEN
      INTEGER OLDCOL,OLDLEV,OUTST,STARTC,STARTL,TENTAT
C BLANK, A CONSTANT EQUAL TO THE CHARACTER BLANK.
C COLUMN, THE POSITION IN CHARB OF THE LAST CHARACTER OF THE SYMBOL
C   BEING PROCESSED.
C INDENT, THE NUMBER OF BLANKS TO BE USED TO INDENT THE CURRENT LINE.
C   IT IS EQUAL TO MIN(INDFAC*STARTL,MAXINT).
C INDFAC, A CONSTANT EQUAL TO THE NUMBER OF COLUMNS TO INDENT PER LEVEL,
C   FOR EXAMPLE 2.
C LEVELL, EQUAL TO MAX(STARTL,LEVELB(TENTAT)).
C LINEL, LENGTH OF THE LONGEST POSSIBLE OUTPUT LINE INCLUDING
C   INDENTATION, FOR EXAMPLE 120.
C MAXINT, THE MAXIMUM NUMBER OF LEADING BLANKS THAT WILL BE PERMITTED
C   FOR INDENTATION, FOR EXAMPLE 50. IT IS REQUIRED THAT MAXINT .LT.
C   LINEL.
C MAXLEN, THE MAXIMUM NUMBER OF NONINDENTATION CHARACTERS FOR THE
C   CURRENT LINE. IT IS EQUAL TO LINEL-INDENT.
C OLDCOL, THE POSITION IN CHARB OF THE LAST CHARACTER OF THE PREVIOUSLY
C   PROCESSED SYMBOL.
```

```
C OLDLEV, THE LEVEL OF THE PREVIOUS SYMBOL.
C OUTST, THE STATE OF OUTSYM. IT IS USED TO GIVE THE EFFECT OF
C   COROUTINES. IT IS INITIALIZED BY THE PARSER TO ZERO. THE STATES ARE
C CODED AS FOLLOWS.
C   0 = INITIALIZATION STATE. THE ROUTINE STARTS IN THIS STATE AND GOES
C     INTO IT AFTER OUTPUTING EACH LINE.
C   1 = INITIAL INCREASING SEQUENCE STATE. THE ROUTINE GOES INTO THIS
C     STATE WHEN THE SECOND SYMBOL OF THE LINE IS PROCESSED AND STAYS IN
C     IT UNTIL OLDLEVEL .GT. MAXLEN OR OLDLEVEL.GE.LEVEL.
C   2 = LEVEL SEGMENT. THE ROUTINE GOES INTO THIS STATE FROM STATE 1 AS
C     SOON AS A SYMBOL WITH A LEVEL NUMBER LESS THAN THAT OF THE
C     PREVIOUS SYMBOL IS FOUND. IT STAYS IN THIS STATE UNTIL A SYMBOL
C     WHICH WILL NOT FIT ON THE CURRENT LINE IS ENCOUNTERED OR UNTIL IT
C     DECIDES TO OUTPUT THE CURRENT LINE.
C   3 = NEXT LINE.THE ROUTINE GOES INTO THIS STATE FROM STATE 1 OR 2
C     WHEN A SYMBOL THAT WILL NOT FIT ON THE CURRENT LINE IS FOUND. IT
C     STAYS IN THIS STATE UNTIL IT DECIDES TO OUTPUT THE CURRENT LINE.
C     THIS STATE IS USED TO DECIDE WHEN SOME OF THE SYMBOLS THAT COULD
C     GO THAT THE END OF THE CURRENT LINE WOULD DO BEST AS PART OF THE
C     NEXT LINE.
C STARTC, THE POSITION IN CHARB OF THE LAST CHARACTER OF THE FIRST
C   SYMBOL OF THE LINE.
C STARTL, THE LEVEL OF THE FIRST SYMBOL OF THE LINE. IT IS EQUAL TO
C   LEVELB(STARTC).
C TENTAT, THE TENTATIVE END OF THE CURRENT LINE. IN STATES 1 AND 2
C   TENTAT ONLY INCREASES. IN STATE 3 IT ONLY DECREASES. INITIALLY
C   TENTAT SHOULD BE ZERO. AFTER THAT ONE HAS STARTC.LE.TENTAT.LE.
C   MAXLEN. SETTING TENTAT AND THEN PRINTING THE LINE IS, OF COURSE, THE
C   MAIN FUNCTION OF THE ROUTINE.
      INTEGER I,JDELTA,LEV
C     PROCESS NEW CHARACTER.
      DELTA = LASTC-LASTS
      LEV = LEVELB(LASTS)
      COLUMN = LASTS
      IF (NEWLIN.GT.0) GO TO 10
C     CHECK STATE.
1     GO TO (2,3,9,17),OUTST
C       INITIALIZATION STATE.
2       TENTAT = COLUMN
        STARTC = COLUMN
        LEVELL = 0
        STARTL = LEV
        INDENT = MIN0(INDFAC*STARTL,MAXINT)
        MAXLEN = LINEL-INDENT
        OUTST = 2
        GO TO 6
C           INITIAL INCREASING SEQUENCE STATE.
3           IF (OLDCOL.GT.MAXLEN) GO TO 16
            IF (LEV.LE.OLDLEV) GO TO 4
              LEVELL = OLDLEV
              GO TO 6
4             OUTST = 3
C         LEVEL SEGMENT STATE.
5         OLDLEV = LEV
          LEVELL = OLDLEV
```

```
          TENTAT = OLDCOL
C       GET NEXT SYMBOL.
5       OLDLEV = LEV
        OLDCOL = COLUMN
7         IF (COLUMN.LT.LASTS) GO TO 8
            IF (LASTS.GT.0.AND.LEVEL.EQ.0) GO TO 10
              RETURN
8           COLUMN = COLUMN+1
            LEV = LEVELB(COLUMN)
            IF (LEV.EQ.0) GO TO 7
              GO TO 1
C           MORE OF LEVEL SEGMENT.
0           IF (OLDCOL.GT.MAXLEN) GO TO 16
            IF (LEV.LE.LEVELL) GO TO 5
            IF (OLDLEV.GE.STARTL.OR.LEVEL.LE.OLDLEV) GO TO 6
C       PRINT LINE.
10      J = 0
        IF (NEWLIN.GT.0.AND.TENTAT.LE.0) TENTAT = LASTS
        IF (TENTAT.LE.0) GO TO 15
          INDENT = MAX0(1,MIN0(INDENT,LINEL-TENTAT))
          TENTAT = MIN0(TENTAT,LINEL-INDENT)
          WRITE (6,11) (BLANK, I = 1,INDENT),(CHARB(I), I=1,TENTAT)
11        FORMAT (120A1)
          IF (TENTAT.GE.LASTC) GO TO 15
          I = TENTAT
C           SHIFT DOWN BUFFER AND REMOVE LEADING BLANKS.
12          I = I+1
            IF (CHARB(I).EQ.BLANK) GO TO 14
13            J = J+1
              CHARB(J) = CHARB(I)
              LEVELB(J) = LEVELB(I)
              I = I+1
              IF (I.LE.LASTC) GO TO 13
            GO TO 15
14          IF (I.LT.LASTC) GO TO 12
15      LASTC = J
        LASTS = LASTC-DELTA
        COLUMN = 0
        TENTAT = 0
        OUTST = 1
        GO TO 7
            NEXT LINE.
16          OUTST = 4
            GO TO 18
17          IF (OLDCOL-TENTAT.GT.
     1        LINEL-MIN0(INDFAC*LEVELB(TENTAT),MAXINT)) GO TO 10
18        IF (LEV.LT.LEVELL) GO TO 19
            IF (OLDLEV.GE.STARTL.OR.LEV.LE.OLDLEV) GO TO 6
              GO TO 10
19          J = TENTAT
20            I = J
              IF (I.LE.STARTC) GO TO 10
21              J = J-1
                IF (LEVELB(J).EQ.0) GO TO 21
              IF (LEVELB(I).GE.LEVELL.OR.LEVELB(I).GE.LEVELB(J)) GO TO 20
```

```
      IF (OLDCOL-I.GT.LINEL-MIND(INDFAC*LEVELB(I),MAXINT))
     1   GO TO 10
      TENTAT = J
      LEVELL = LEVELB(I)
      IF (LEVELL.GT.LEV) GO TO 20
    GO TO 6
 END
```

```
begin integer n; read(n); begin int
    1       2 32    23332     2

eger i; real svalue,dvalue,pnorm,qn
   3 43    3       44     44    44

orm,scalar; array x,y[1:n]; for i :
  44     43     3 444444443   3 4

= 1 step 1 until n do read(x[i]); f
4 4    4 4     4 4  4    45555553

or i := 1 step 1 until n do read(y[
 3 4   4 4    4 4     4 4  4    4555

i]); svalue := dvalue := 0; begin a
5553      3  4       4  4 43     3

rray s[1:n]; scalar := 0; for i :=
   4 5555554      4  5 54   4 5  5

1 step 1 until n do begin s[i] := x
5    5 5     5 5  5     5 6777  7 7

[i]+y[i]; scalar := scalar+s[i]×x[i
777777776      6  7      7777777777

]; svalue := svalue+s[i]×s[i] end;
76      6  7      77777777777   54

pnorm := scalar/svalue; for i := 1
    4  5      55     54   4 5  5 5

step 1 until n do print(s[i]×pnorm)
   5 5     5 5  5     5666666    66

 end; begin array d[1:n]; scalar :=
   33     3     4 5555554      4  5

 0; for i := 1 step 1 until n do be
54   4 5  5 5    5 5     5 5  5

gin d[i] := x[i]-y[i]; scalar := sc
  5 6777  7 7777777776      6  7

alar+d[i]×x[i]; dvalue := dvalue+d[
   777777777776      6  7      7777

i]×d[i] end; qnorm := scalar/dvalue
7777777   54     4  5      55     5

; for i := 1 step 1 until n do prin
4   4 5  5 5    5 5     5 5  5

t(d[i]×qnorm) end; print(abs(pnorm×
5666666    66   33     34  44    44

sqrt(svalue)-qnorm×sqrt(dvalue))) e
   44     444    44   44     4444

nd end
 2   1
```

Figure 1
The sample program with our identation and with the indentation level for each symbol.

```
begin integer n; read(n); begin
    integer i; real svalue,dvalue
      ,pnorm,qnorm,scalar;
    array x,y[1:n]; for i := 1
      step 1 until n do read(x[i]
        );
    for i := 1 step 1 until n do
      read(y[i]);
    svalue := dvalue := 0; begin
      array s[1:n]; scalar := 0;
      for i := 1 step 1 until n
        do begin s[i] := x[i]+y[i
            ];
          scalar := scalar+s[i]×x
            [i];
          svalue := svalue+s[i]×s
            [i] end;
    pnorm := scalar/svalue; for
      i := 1 step 1 until n do
      print(s[i]×pnorm) end;
  begin array d[1:n]; scalar :=
      0;
    for i := 1 step 1 until n
      do begin d[i] := x[i]-y[i
          ];
          scalar := scalar+d[i]×x
            [i];
          dvalue := dvalue+d[i]×d
            [i] end;
      qnorm := scalar/dvalue; for
        i := 1 step 1 until n do
        print(d[i]×qnorm) end;
    print(abs(pnorm×sqrt(svalue)-
      qnorm×sqrt(dvalue))) end
end
```

Figure 2

The sample program indented with the basic algorithm.

```
begin integer n; read(n); begin
    integer i; read svalue,dvalue
      ,pnorm,qnorm,scalar;
    array x,y[1:n]; for i := 1
      step 1 until n do
      read(x[i]);
    for i := 1 step 1 until n do
      read(y[i]);
    svalue := dvalue := 0; begin
      array s[1:n]; scalar := 0;
      for i := 1 step 1 until n
        do begin
          s[i] := x[i]+y[i];
          scalar := scalar+s[i]×x
            [i];
          svalue := svalue+s[i]×s
            [i] end;
      pnorm := scalar/svalue; for
        i := 1 step 1 until n do
        print(s[i]×pnorm) end;
    begin array d[i:n];
      scalar := 0; for i := 1
        step 1 until n do begin
          d[i] := x[i]=y[i];
          scalar := scalar+d[i]×x
            [i];
          dvalue := dvalue+d[i]×d
            [i] end;
      qnorm := scalar/dvalue; for
        i := 1 step 1 until n do
        print(d[i]×qnorm) end;
    print(abs(pnorm×sqrt(svalue)-
      qnorm×sqrt(dvalue))) end
end
```

Figure 3
The sample program indented with the end fix algorithm.

```
begin
  integer n;
  read(n);
  begin
    integer i;
    real
      svalue,dvalue,pnorm,qnorm,
      scalar
    ;
    array x,y[1:n];
    for
      i := 1 step 1 until n do
      read(x[i])
    ;
    for
      i := 1 step 1 until n do
      read(y[i])
    ;
    svalue := dvalue := 0;
    begin
      array s[1:n];
      scalar := 0;
      for
        i := 1 step 1 until n do
        begin
          s[i] := x[i]+y[i];
          scalar
            := scalar+s[i]×x[i]
          ;
          svalue
            := svalue+s[i]×s[i]
        end
      ;
      pnorm := scalar/svalue;
      for
        i := 1 step 1 until n do
        print(s[i]×pnorm)
    end;
    begin
      array d[1:n];
      scalar := 0;
      for
        i := 1 step 1 until n do
        begin
          d[i] := x[i]-y[i];
          scalar
            := scalar+d[i]×x[i]
          ;
          dvalue
            := dvalue+d[i]×d[i]
        end
      ;
      qnorm := scalar/dvalue;
      for
        i := 1 step 1 until n do
        print(d[i]×qnorm)
    end;
    print
      (abs(pnorm×sqrt(svalue)-
      qnorm×sqrt(dvalue)))
  end
end
```

Figure 4
The sample program indented with the short same level algorithm.

```
begin integer n; read(n);
  begin integer i;
    real svalue,dvalue,pnorm,
      qnorm,scalar;
    array x,y[1:n];
    for i := 1 step 1 until n do
      read(x[i]);
    for i := 1 step 1 until n do
      read(y[i]);
    svalue := dvalue := 0;
    begin array s[1:n];
      scalar := 0;
      for i := 1 step 1 until n
        do
        begin s[i] := x[i]+y[i];
          scalar := scalar+s[i]×x
            [i];
          svalue := svalue+s[i]×s
            [i] end;
      pnorm := scalar/svalue;
      for i := 1 step 1 until n
        do print(s[i]×pnorm) end;
    begin array d[i:n];
      scalar := 0;
      for i := 1 step 1 until n
        do
        begin d[i] := x[i]-y[i];
          scalar := scalar+d[i]×x
            [i];
          dvalue := dvalue+d[i]×d
            [i] end;
    qnorm := scalar/dvalue;
    for i := 1 step 1 until n
      do print(d[i]×qnorm) end;
  print(abs(pnorm×sqrt(svalue)-
    qnorm×sqrt(dvalue))) end
  end
```

Figure 5

The sample program indented with the level end algorithm or with the shallow break algorithm.

```
begin integer n; read(n);
  begin integer i;
    real svalue,dvalue,pnorm,
      qnorm,scalar
    ;
    array x,y[1:n];
    for i := 1 step 1 until n do
      read(y[i]);
    svalue := dvalue := 0;
    begin array s[1:n];
      scalar := 0;
      for i := 1 step 1 until n
        do
        begin S[i] := x[i]+y[i];
          scalar := scalar+s[i]×x
            [i]
          ;
          svalue := svalue+s[i]×s
            [i]
        end
      ;
      pnorm := scalar/svalue;
      for i := 1 step 1 until n
        do print(s[i]×pnorm)
    end;
    begin array d[1:n];
      scalar := 0;
      for i := 1 step 1 until n
        do
        begin d[i] := x[i]-y[i];
          scalar := scalar+d[i]×x
            [i]
          ;
          dvalue := dvalue+d[i]×d
            [i]
        end
      ;
      for i := 1 step 1 until n
        do print(d[i]×qnorm)
    end;
    print(abs(pnorm×sqrt(svalue)-
      qnorm×sqrt(dvalue)))
  end
end
```

Figure 6
The sample program indented with the shallow break with explicit level ends algorithm.

References

1. McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," CACM 1960 (3) p. 184-195.

2. Naur, Peter, et. al., "Report on the Algorithm Language ALGOL 60," CACM 1960 (3) p. 299-314.

3. McCracken, D.D., A Guide to FORTRAN Programming, Wiley (New York, 1961) p. 9.

4. Prager, William, Introduction to Basic FORTRAN Programming and Numerical Methods, Blaisdell (New York 1965) p. 26.

5. Organick, E.L., A FORTRAN IV Primer, Addison-Welsey (Reading Mass 1966) p. 97. Compare with A FORTRAN Primer, 1963 by same author and publisher.

6. Lowry, S.E. and C.W. Medlock, "Object Code Optimization," CACM 1969 (12) p. 13-22.

7. Baumann, R., M. Feliciano, F.L. Bauer and K. Samelson, "Introduction to ALGOL," Prentice-Hall (Englewood Cliffs, 1964) p. 45-46.

8. Lewis, P.M. and R.E. Stearns, "Syntax-directed Transduction," JACM 1968 (15) p. 465-488.

9. Knuth, D.E., "On the Translation of Languages from Left to Right," Information and Control, 1965 (8) p. 607-639.

10. Baker, B.S., "An Algorithm for Structuring Programs," third ACM Symposium on Principles of Programming Languages, 1976, p. 113-124.

11. Wise, D.S., D.P. Friedman, S.C. Shapiro, and Mitchell Wand, "Boolean-Valued Loops," BIT 1975 (15), p. 431-451.