Computer Science Department

TECHNICAL REPORT

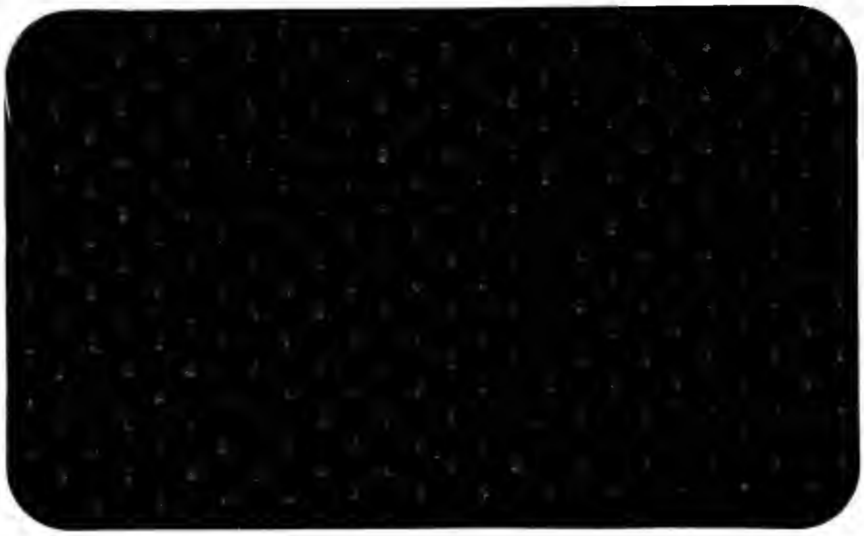
COMPILE-TIME ANALYSIS OF DATA LIST-FORMAT LIST CORRESPONDENCES

By
Paul Abrahams
and
Lori Clarke

February 1979
Report No. 010

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



COMPILE-TIME ANALYSIS OF DATA LIST-FORMAT LIST CORRESPONDENCES

By
Paul Abrahams
and
Lori Clarke

February 1979
Report No. 010

Compile-Time Analysis of Data List-Format List Correspondences

Paul Abrahams*
Computer Science Department
New York University

and

Lori Clarke**
Computer and Information Science Department
University of Massachusetts

*This research was supported in part by the U.S. Department of Energy under contract EY-76-C-03-3077, and in part by Control Data Corporation under contract C08AA.

**This research was supported in part by AFOSR under contract AFOSR-77-3287, and in part by the National Science Foundation under grant MCS77-02101.

ABSTRACT

Formatted input-output is available in a number of programming languages. In the most general case, the correspondence between data items and format items cannot be determined during compilation, and so it is determined dynamically during execution. However, in most pairs of data and format lists that occur in practice, determination of the correspondence is in fact possible during compilation. Although some commercial compilers make this determination, there is little published literature on the subject. In this paper, we briefly examine three areas in which compile-time determination of the data-format correspondence is useful: optimization, program validation, and automatic test data generation. A formalism for stating the problem is given, and a solution is discussed in terms of formal language theory. Using this formalism, an algorithm for determining the correspondence is given, and its application is illustrated by examples in both PL/I and FORTRAN.

Keywords and key phrases: formats, compilers, program optimization, program validation, test data generation, input-output, static program analysis.

1. Introduction

Formatted input-output plays an important role in FORTRAN and PL/I, and is also provided in Algol 68 and in certain Algol extensions. A formatted input-output operation is specified by providing a data list and a format list. The data list specifies the items to be read or written, while the format list specifies how the items are represented on the input or output medium. In most cases that occur in practice, it is possible to determine during compilation how data items are paired with format items. That isn't too surprising, since the programmer should have anticipated the correspondence when the data list and format list were composed. Consolidating the two lists into one list of pairs eliminates the need for expensive execution-time linkage mechanisms, and moreover makes it possible to derive information useful in program validation and in automatic test data generation. In this paper, we present an algorithm for converting the two lists into a single list of pairs. Although the conversion is trivial if the lists are expanded by writing out all iterations in full, it is not trivial if we desire to retain as much of the original iteration structure as possible, which our algorithm does.

In some cases, our algorithm rejects the input because the correspondence cannot be determined until execution. For example, consider the FORTRAN statements:

```
        WRITE (1,5) (A(I),I = 1,M),B
    5   FORMAT (E14.3,E12.3)
```

We cannot know prior to execution whether B will correspond to the format E14.3 or to the format E12.3, since that depends on whether M is even or odd.

In this paper, we shall consider the application of our algorithm to Fortran and PL/I; we have not attempted to apply it to other languages. The current status of our work is that the algorithm has been programmed (in SNOBOL) and tested, but it has not been incorporated into an actual compiler. Although the SNOBOL implementation is effective for testing and experimentation, a practical implementation would necessarily use lists rather than character strings as its underlying representation.

2. Applications

The major application of our algorithm is the optimization of formatted input-output. Ordinarily the execution of a formatted input-output statement is implemented by a pair of coroutines, one for the data list and one for the format list. Each coroutine keeps track of the position in the list, and finds the next item when it is called. Control shuttles back and forth between the two routines, and when a data-format pair is obtained, the appropriate input or output action is taken. The code required for this conversation can be eliminated if the correspondence is known in advance, since then the proper format can be compiled directly into the data list. Knuth's study of FORTRAN programs [10] found that about 25% of the overall execution time was spent in the I/O editing routines. Therefore, we expect compile-time analysis to produce a noticeable reduction in execution time.

The correspondence between the data list and format list can be used in program validation to detect certain types of programming errors. For instance, we can check whether the type of each data item agrees with the type of the corresponding format item. When making this check, we would want to ignore certain distinctions among format items. For instance, in the example given earlier, the formats E14.3 and E12.3 would be treated as one and the same since they both match variables of type REAL. Since formats are a major source of errors for beginning FORTRAN programmers, this check would be valuable in diagnostic FORTRAN compilers. In PL/I, however, all printable data types convert to all other printable data types, so formats are always correct from this viewpoint.

Once the correspondence between a data item and a format item is known, then a range of permissible values for the data item is also known. This information can be useful for more sophisticated validations. For example, suppose we have the FORTRAN sequence:

```
         WRITE (5,10) N
     10  FORMAT (I2)
```

A warning should be issued if N is potentially greater than 99 or less than -9. Recent work in static program analysis [1,4,5] should be useful in this type of validation.

In automatic test data generation, a topic investigated by Clarke in [3], an attempt is made to determine legal input data that will exercise particular program paths. Knowing the format specifications of a data item determines a range of potential values for the data item. This in turn may limit the range of other variables. For example, in the following FORTRAN sequence:

```
         READ (5,10) I,J
     10  FORMAT(I2,I10)
         K = I + J
```

we note that the possible range of legal values for I is -9 to 99 and for J is -99 to 999, while the possible range of values for K is -108 to 1098.

Current test data generation systems have ignored format information, even though FORTRAN and PL/I have been the languages most frequently analyzed by such systems [3,7,9,12,13]. Of course, it is possible to compare the generated data with the corresponding format statements, using the coroutines approach mentioned in connection with optimization. However, if the generated data is not consistent with the corresponding format, an expensive reanalysis is necessary. It would be more economical to extract the data-format correspondence before analysis.

The optimization aspect of formatted input-output is touched upon by Lee in his FORTRAN-oriented book on compiler writing [11]. Moreover, the IBM PL/I Optimizing Compiler [8] does match data lists with format lists. However, the circumstances under which this matching is done, and the method used to accomplish it, are proprietary.* Although other commercial compilers also perform this matching, we are not aware of any published literature about them. Torsun and Robinson [14] have developed a system that preprocesses formats, but their system does not perform any compile-time analysis on data lists that contain iterations. Their discussion deals mostly with the numerical encoding of formats, and has little to say about the problems considered here.

3. Notation

From now on, we will refer to format items as F-items and to data items as D-items. Similarly, a format list will be referred to as an

*We wish to emphasize that the methods developed in this paper were devised without any knowledge of the IBM method, as neither of us has access to it.

F-list, and a data list as a D-list. We distinguish three kinds of repetition factors: constant, variable, and infinite. Constant repetition factors are written explicitly. Variable repetition factors are denoted by V_1, V_2, ... V_s and infinite repetition factors by ∞. Essentially the same formation rules, but with different individual items, can be used for D-lists and F-lists:

(1) An individual item is a component.

(2) If $x_1,x_2,...,x_k$ are components, then $[x_1,x_2,...,x_k]$ is a nonrepeated sequence with subcomponents $x_1,...,x_k$.

(3) If $x_1,x_2,...,x_k$ are components and r is a constant, variable or infinite repetition factor, then $r[x_1,x_2,...,x_k]$ is a repeated sequence with subcomponents $x_1,x_2,...,x_k$. A repeated sequence is a component.

(4) If $[x_1,x_2,...,x_k]$ is a nonrepeated sequence whose individual items are all D-items, then $[x_1,x_2,...,x_k]$ is a D-list. If $[x_1,x_2,...,x_k]$ is a nonrepeated sequence whose individual items are all F-items, then $[x_1,x_2,...,\infty[x_k]]$ is an F-list.

These rules require that sequences always appear with repetition factors except at the outermost level. Thus, an individual item with a repetition factor must be replaced by a unit list with that repetition factor (e.g., we replace $4F_1$ by $4[F_1]$). The asymmetry in rule (4) is accounted for by the facts that infinite repetition cannot occur in D-lists (except by error in certain PL/I situations) and that any F-item following an infinite repetition can just as well be ignored. For FORTRAN and PL/I, there are further restrictions on F-lists. In FORTRAN, no variable repetition factor can occur in the F-list and only the rightmost level-one parenthesized list has an implied infinite repetition factor. In PL/I, infinite repetition can be applied only to the entire F-list, so that k must be 1 in rule (4).

Some examples are in order to show how the notation corresponds to reality. Consider the FORTRAN example:

```
      WRITE (5,100) ((A(I,J), J = 1,7), I, I = 1,M)
  100 FORMAT (7E10.1, I3, (7E12.2,I2))
```

The D-list and F-list are then:

$$[V_1[7[D_1],D_2]]$$

and

$$[7[F_1],F_2,\infty[7[F_3],F_4]]$$

respectively, with

D_1 = A(I,J), D_2 = I, F_1 = E10.1, F_2 = I3, F_3 = E12.2, F_4 = I2, V_1 = M.

We have chosen to treat A(I,J) as a single item, although for certain applications of test data generation, a finer distinction may be desirable. A similar example in PL/I is:

```
PUT EDIT (((A(I,J) DO J = 1 TO 7), I DO I = 1 TO M))
          ((7)E(10,1), F(3), (M-1) ((7)E(12.2), F(2)))
```

The D-list is represented as in the FORTRAN example, but the F-list is

$$[\infty[7[F_1],F_2,V_2[7[F_3],F_4]]]$$

where V_2 = (M-1) and the other symbols are the same as before.

4. <u>The Correctness Problem in Terms of Formal Language Theory</u>

Provided that the F-list contains no variable repetition factors, the correctness problem can be shown to be solvable using results from formal language theory. If variable repetition factors are present in the F-list, and nothing is known about them, then formal language theory is of no help. For consider:

D-list: $[V_1[D_1],D_2]$

F-list: $[V_2[F_1],F_2]$

Assume moreover that F_1 and F_2 are valid formats for D_1 and D_2 respectively. Even though the two lists have the same form, we cannot tell whether they match correctly.

If there are no variable repetition factors in the F-list, then we can transform both lists into regular expressions (see, for instance, Hopcroft and Ullman [6]) as follows:

(1) If $D_{i_1},D_{i_2},\ldots,D_{i_j}$ are the D-items that match F_j, then replace F_j by the expression

$$(D_{i_1} \vee D_{i_2} \vee \ldots \vee D_{i_j})$$

(2) Replace each variable or infinite repetition factor by *, indicating zero or more occurrences.

(3) Expand out each constant repetition factor.

We then have two regular languages, $\underline{D}$ and $\underline{F}$ respectively, for the D-list and F-list. We then see:

(1) If $\underline{D} \subset \underline{F}$, then the correspondence is valid.

(2) If $\underline{D} \cap \underline{F}$ is empty, then the correspondence cannot be valid.

(3) In all other cases, the validity of the correspondence cannot be determined.

These statements follow from the observation that the sentences in $\underline{D}$ are all the possible sequences of D-items, while the sentences in $\underline{F}$ are obtained by taking all the possible sequences of F-items (a necessarily infinite set) and replacing each F-item by all possible D-items that it can match. Now the relation between D and F can be algorithmically determined since the containment and intersection problems for regular languages are solvable (again, see Hopcroft and Ullman). It follows that the correctness problem is indeed solvable. Since FORTRAN has no variable repetition factors in its formats, the correctness problem can be solved for that language, in the sense that we can determine which of the three cases given above is applicable. For PL/I, it cannot be solved except for formats having constant repetition factors.

Although formal language theory shows that the correctness problem is solvable, and even provides an algorithm for solution, that algorithm is not a practical one. The formal solution requires that all constant repetitions be fully expanded, and moreover requires that we construct the product of two finite-state machines and then test the language defined by the product for emptiness. For a practical algorithm, we use the same methods as we use for the other applications, and actually find the correspondence between data items and format items.

5. Method of Solution

A solution to the correspondence problem can be expressed by replacing each D_i in a data list by a pair $<D_i, F_j>$, where F_j is the format that matches D_i. First, we define the inner cardinality of a repeated sequence to be the number of individual items in the immediately contained nonrepeated sequence, with repetitions counted. For instance, the inner cardinality of $3[2[D_1],5[D_2]]$ is 7. The inner cardinality is variable if the sequence contains any variable repetition factors. The inner cardinality can be computed in an obvious way by analyzing nested repeated sequences from the inside out.

We present the algorithm as a sequence of operations, using a semiformal style of English adopted from the recent PL/I standard [2]. The algorithm is executed by performing the operation match, whose inputs are a D-list and an F-list, and whose output is a DF-list, i.e., a list of pairs. The algorithm proceeds by a sequence of reductions. When both the D-list and the F-list begin with a single item, we can remove those items from the two lists and construct a new item for the DF-list. Moreover, if both the D-list and the F-list start with a repeated sequence, and the two sequences both have the same repetition factor and the same inner cardinality, then we can add a corresponding repeated sequence to the DF-list, applying match recursively to obtain the inner nonrepeated sequence. (It is this recursion that enables us to retain most of the iterative structure of the original lists.) The rest of the algorithm is concerned with modifying the D-list and the F-list so as to get them into a form in which the initial components can be paired up as we have just described.

In certain cases, when variable repetition factors are encountered, the correspondence between the D-list and the F-list cannot be determined until execution. In these cases, the algorithm rejects the input. To see that variable repetition factors can cause this difficulty, consider the case:

D-list: $[V_1[D_1],D_2]$

F-list: $[F_1,F_2]$

This case is a translation of the example given in the Introduction; the proper pairing of D_2 depends on the value of V_1. On the other hand, some cases involving variable repetition factors can be treated. For instance, the pair:

D-list: $[V_1[D_1]]$

F-list: $[\infty[F_1]]$

yields the DF-list $[V_1[<D_1,F_1>]]$.

match(ds,fs)

where ds is a D-list and fs is an F-list

Result: a DF-list

Note: fs will always include at least as many items as ds.

Step 1. Let dfs be an empty list.

Step 2. Perform Step 2.1 repeatedly until ds is empty. Then return dfs as the value of match.

Step 2.1. Let cde and cfe be, respectively, the first component of ds and of fs.

Case 2.1.1. cde and cfe are both individual items.

Append the pair <cde,cfe> to dfs. Delete cde from ds and delete cfe from fs.

Example: ds = $[D_1,2[D_2]]$

fs = $[F_1,\infty[F_2]]$

new pair = $<D_1,F_1>$

Case 2.1.2. Either cde or cfe is an individual item, while the other is a repeated sequence with a constant or infinite repetition factor. If cfe is the individual item, perform split (1,ds) to obtain a new ds. Otherwise perform split (1,fs) to obtain a new fs.

Example: ds = $[D_1, 2[D_2]]$

fs = $[6[F_1], \infty[F2]]$

new fs = $[F_1, 5[F_1], \infty[F_2]]$

Case 2.1.3. Either cde or cfe is an individual item, while the other has a variable repetition factor.

The input is rejected.

Example: ds = $[V_1[D_1]]$

fs = $[F_1, \infty[F_2]]$

Note that in this example, V_1 may or may not be greater than 0.

Case 2.1.4. cde and cfe are both repeated sequences with the same inner cardinality.

Let rd and rf be the repetition factors of cde and cfe respectively.

Case 2.1.4.1. rd and rf are identical.

Let nsd and nsf be the nonrepeated sequences in cde and cfe respectively. Perform match(nsd,nsf) to obtain a DF-list, dfl. If rd is one, then append dfl to dfs; otherwise, append rd[dfl] to dfs. Delete cde and cfe from ds and fs respectively.

Example: ds = $[5[D_1, D_2]]$

fs = $[5[2[F_1]], \infty[F_2]]$

new component of DF-list = $5[<D_1, F_1>, <D_2, F_1>]$

Case 2.1.4.2. rd and rf are different constants, or rf is infinite.

If rd < rf, perform split(rd,fs) to obtain a new fs.

Otherwise, perform split (rf,ds) to obtain a new ds.

Example: ds = $[4[D_1], D_2]$

fs = $[\infty[F_1]]$

new fs = $[4[F_1], \infty[F_1]]$

Case 2.1.4.3. rd is variable and rf is infinite.

Perform split(rd,fs) to obtain a new fs.

Example: ds = $[V_1[D_1]]$

fs = $[\circ[F_1]]$

new fs = $[V_1[F_1],\infty[F_1]]$

Case 2.1.4.4. (Otherwise.)

The input is rejected.

Example: ds = $[V_1[D_1]]$

fs = $[3[F_1],\infty[F_2]]$

Case 2.1.5. cde and cfe are both repeated sequences with different, but constant, inner cardinalities, nd and nf respectively. Let lcm be the least common multiple of nd and nf, and let md = lcm/nd, mf = lcm/nf.* Let rd and rf be the repetition factors of cde and cfe respectively. Let nr = min(rd/md,rf/mf) if neither rd nor rf is variable, and let nr be undefined otherwise.

Case 2.1.5.1. nr is defined and nr > 1.

Step 2.1.5.1.1. If rd > nr*md, perform split (nr*md,ds) to obtain a new ds. If rf > nr*mf, perform split (nr*mf,fs) to obtain a new fs. (nr will be the new repetition factor for the first component both of ds and of fs.)

Note: It is possible that zero, one or two split operations will be performed in this step.

Step 2.1.5.1.2. If md > 1, replace the first component of ds by nr[md[s]], where s is the nonrepeated sequence of cde.

*We use "/" to indicate integer division with the remainder discarded.

Step 2.1.5.1.3. If mf > 1, replace the first component of fs by nr[mf[s]], where s is the nonrepeated sequence of cfe.

Note: On the next step, Case 2.1.4.1. will apply, since both ds and fs will start with a component having repetition factor nr and inner cardinality lcm.

Example: ds = $[8[D_1,D_2]]$

fs = $[\infty[2[F_1],F_2]]$

nd=2, nf=3, lcm=6, md=3, mf=2, rd=8, rf=∞, nr=2

new ds = $[2[3[D_1,D_2]],2[D_1,D_2]]$

new fs = $[2[2[2[F_1],F_2]],\infty[2[F_1],F_2]]$

Case 2.1.5.2. nr is defined and nr ≤ 1.

If nd > nf, perform split(1,ds); otherwise perform split(1,fs).

Note: In this case, one or both of the first components of ds and fs contains too few elements to allow us to extract a common repeated part, so we expand the longer one. If necessary, the shorter one will be expanded on the next iteration.

Example: ds = $[3[D_1]]$

fs = $[\infty[F_1,4[F_2]]]$

nd=1, nf=5, lcm=5, md=5, mf=1, rd=3, rf=∞, nr=0

new fs = $[F_1,4[F_2],\infty[F_1,4[F_2]]]$

Case 2.1.5.3. rd is variable, rf is infinite, and nd is a multiple k of nf.

Let s be the nonrepeated sequence of cfe. Replace cfe by rd[k[s]],∞[s].

Note: Both ds and fs now start with a component with repetition factor rd and inner cardinality nd.

Example: ds = $[V_1[D_1,D_2]]$

fs = $[\infty[F_1]]$

new fs = $[V_1[2[F_1]],\infty[F_1]]$

Case 2.1.5.4. rd is variable, but Case 2.1.5.3 does not apply.

Reject the input.

Example: ds = $[V_1[D_1]]$

fs = $[\infty[F_1,6[F_2]]]$

Note: Although in practice it may be possible to solve this case, the solution cannot be expressed in our formalism. The solution would be:

$[\infty[<D_1,F_1>,6[<D_1,F_2>]]]$

with an auxiliary test needed to ensure that only V_1 elements are processed.

Case 2.1.6. (Otherwise.)

Reject the input.

split(k,s)

where k is an integer and s is a nonrepeated sequence.

Result: a modified nonrepeated sequence.

Step 1. Let c be the first component of s. c must be a repeated sequence, so it has repetition factor r and contains a nonrepeated seauence cs.

Step 2. Let k2 be r - k. (Note that ∞ minus anything is ∞.)

Replace c by the two components k[cs],k2[cs].

Step 3. If either k or k2 is 1, replace the corresponding component by cs, i.e., delete the repetition factor.

An example of the algorithm applied to a compound case is shown in Figure 1. Two smaller examples, omitting the intermediate steps, are:

Step	ds	fs	dfs	Case
1	$[20[D_1,D_2]]$	$[10[F_1,F_2,F_3],\infty[F_4]]$	λ[+]	2.1.5.1
2	$[5[3[D_1,D_2],5[D_1,D_2]]$	$[5[2[F_1,F_2,F_3]],\infty[F_4]]$	λ	2.1.4.1
2.1	$[3[D_1,D_2]]$	$[2[F_1,F_2,F_3]]$	λ	2.1.5.2
2.2	$[D_1,D_2,2[D_1,D_2]]$	$[F_1,F_2,F_3,F_1,F_2,F_3]$	λ	2.1.1 (2 times)
2.3	$[2[D_1,D_2]]$	$[F_3,F_1,F_2,F_3]$	$[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle]$	2.1.2
2.4	$[D_1,D_2,D_1,D_2]$	$[F_3,F_1,F_2,F_3]$	$[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle]$	2.1.1 (4 times)
3	$[5[D_1,D_2]]$	$[\infty[F_4]]$	$[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle,\langle D_1,F_3\rangle,\langle D_2,F_1\rangle,\langle D_1,F_2\rangle,\langle D_2,F_3\rangle]$	2.1.5.1
4	$[5[D_1,D_2]]$	$[5[2[F_4]],\infty[F_4]]$	$[5[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle,\langle D_1,F_3\rangle,\langle D_2,F_1\rangle,\langle D_1,F_2\rangle,\langle D_2,F_3\rangle]]$	2.1.4.1
4.1	$[D_1,D_2]$	$[2[F_4]]$	$[5[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle,\langle D_1,F_3\rangle,\langle D_2,F_1\rangle,\langle D_1,F_2\rangle,\langle D_2,F_3\rangle]]$	2.1.2
4.2	$[D_1,D_2]$	$[F_4,F_4]$	$[5[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle,\langle D_1,F_3\rangle,\langle D_2,F_1\rangle,\langle D_1,F_2\rangle,\langle D_2,F_3\rangle]]$	2.1.1 (2 times)
5	λ	$[\infty[F_4]]$	$[5[\langle D_1,F_1\rangle,\langle D_2,F_2\rangle,\langle D_1,F_3\rangle,\langle D_2,F_1\rangle,\langle D_1,F_2\rangle,\langle D_2,F_3\rangle],$ $5[\langle D_1,F_4\rangle,\langle D_2,F_4\rangle]]$	

[+] 'λ' indicates an empty list

Figure 1

Example 2

data list:	$[3[5[D_1,D_2],D_3,D_4]]$
format list:	$[4[6[F_1,F_2]],\infty[6[F_1,F_2]]]$
resulting list:	$[3[5[<D_1,F_1>,<D_2,F_2>],<D_3,F_1>,<D_4,F_2>]]$

Example 3

data list:	$[200[D_1,5[D_2],5[D_3]]]$
format list:	$[\infty[F_1,5[F_2],5[F_3]]]$
resulting list:	$[200[<D_1,F_1>,5<D_2,F_2>,5<D_3,F_3>]]$

6. Treatment of Control Formats

The formal model we have presented does not account for control formats, e.g., hollerith fields in FORTRAN formats and skips to the next record. However, control formats are easily accounted for by associating them with data formats. For applications other than optimization, control formats are irrelevant and can be ignored.

In associating control formats with data formats, we must distinguish between control formats that are executed only if a following data format is used, and those that are executed whenever the preceding data format is used. In FORTRAN, the rule is that following control formats are executed unless the end of the entire format is encountered. Thus, if we execute:

```
          WRITE(u,10)A,B
     10   FORMAT(1H1, I5, 1H*, I5, 1H*)
```

both stars are printed. Hence the first I5 has two control formats (1H1 and 1H*) associated with it, while the second I5 has the second 1H* associated with it.

In FORTRAN, we must also account for the peculiar behavior of end-of-line. An end-of-line is generated whenever the right end of the format is encountered. Hence a line skip must be associated as a post-format for the last data format in the list. A line skip also occurs at the completion of the entire operation, unless one has just been produced; this final skip can be generated independently of our algorithm.

In PL/I, control formats are not used unless the following data format is also used. Hence in PL/I, all control formats are associated as pre-formats with data formats.

7. Actual Experience

To demonstrate the effectiveness of the algorithm in determining data-format correspondence, a group of programs were analyzed. Thirteen programs were chosen, all written in FORTRAN. The programs were selected randomly; listings were obtained from the graduate students available one Saturday afternoon. Some of the programs were large and had been coded by numerous people. In all, the programs were the work of about 25 programmers.

Two hundred and fifty-one data and format statements were examined and only fifteen could not be completely analyzed by the algorithm. Thus, this technique failed in only six percent of the cases examined.

A few observations about the formats are also of interest. About 25 percent of the data lists were empty and, thus, the format list had only format control information. About 40 percent of the data and format lists examined could be analyzed completely by using just Case 1 of the algorithm. None of the examined lists were as complex as those presented in the previous examples; none required the use of Case 2.1.5. Though PL/I programs were not analyzed, we have no reason to believe the results would be substantially different.

Acknowledgement

We wish to thank the referees for several helpful criticisms of an earlier draft of this paper.

References

[1] F.E. Allen, and J. Cocke, "A Program Data Flow Analysis Procedure," CACM, 19,3, March 1976, pp. 137-147.

[2] American National Standards Institute, "American National Standard: Programming Language PL/I," ANSI X3.53-1976.

[3] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Software Engineering, Vol. SE-2, Sept. 1976, pp. 215-222.

[4] L.D. Fosdick, and L.J. Osterweil, "Data Flow Analysis in Software Reliability," Vol. 8-3, Sept. 1976, pp. 305-330.

[5] W.H. Harrison, "Compiler Analysis of the Value Ranges for Variables," IEEE Trans. Software Engineering, Vol. SE-3, May 1977, pp. 243-250.

[6] J. Hopcroft, and J. Ullman, "Formal Languages and Their Relation to Automata," Addison-Wesley, Reading, Mass., 1969.

[7] W.E. Howden, "Methodology for the Generation of Program Test Data," IEEE Trans. Comput., Vol. C-24, May 1975, pp. 554-559.

[8] IBM Corporation, "OS PL/I Checkout and Optimizing Compilers: Language Reference Manual," Order Number GC33-0009-3.

[9] J.C. King, "A New Approach to Program Testing," in Proc. Int. Conf. Reliable Software, April 1975, pp. 228-233.

[10] D.C. Knuth, "An Empirical Study of FORTRAN Programs," Software-Practice and Experience, Vol. 1, 1971, pp. 105-133.

[11] J.A.N. Lee, "The Anatomy of a Compiler," 2 ed., Van Nostrand Reinhold, New York, 1975.

[12] E.F. Miller, and R.A. Melton, "Automated Generation of Test Case Datasets," in Proc. Int. Conf. Reliable Software, April 1975, pp. 51-58.

[13] C.V. Ramamoorthy, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data," IEEE Trans. Software Engineering, Vol. SE-2, Dec. 1976, pp. 293-300.

[14] I. Torsun, and S. Robinson, "Non-'Interpretive' FORTRAN Input/Output," Software-Practice and Experience, 7(2), March-April 1977, pp. 205-213.