THIRD GENERATION COMPILER DESIGN

Marvin V. Zelkowitz
Department of Computer Science, University of Maryland
College Park, Maryland

Compilers, besides testing for errors in a particular implementation of an algorithm, can be implemented to analyze program structure. This information can be fed back to the programmer in order to improve the structure, reliability and efficiency of the resulting program. This paper surveys several techniques that are currently implementable in a compiler, describes several new techniques that can be applied to programs, and briefly describes one such implementation of many of these ideas.

1. INTRODUCTION

The development of reliable software is currently proceeding along several paths. Languages are being developed which a priori result in correct, more understandable and more manageable programs [8]. At the same time others are developing proof techniques that, a posteriori, show that a program is correct [5]. A third path is the development of techniques that result in information during the development phase of a program being fed back to the programmer in order to suggest changes to be made in the source program [15].

Compilers, as an example of this third approach, seem to be entering into a third phase of development since they first appeared some twenty years ago. The first compilers (and unfortunately still the dominant class) simply converted a source language program into an equivalent machine language program. If there were any syntax errors in the program, then the compiler generated an error message and terminated the translation process. Semantic errors were usually not detected by the compiler and thus caused the program to give unpredictable results during program execution.

The second class of compiler first appeared during the early 1960's. These compilers of the load and go diagnostic class attempted to aid in program development [2,3,16,17]. Should there have been an obvious error in syntax, then the compiler would generate an error message, "fix" the error, and continue compilation. The code generated also detected as many execution errors as possible; thus many semantic errors were caught during program execution.

While diagnostic compilers are very useful in fixing errors in a particular implementation of an algorithm - once an error has been detected - questions such as the reliability or efficiency of the resulting program are not addressed. It is not possible to detect whether a program is "good" or "poor"; only that it produces correct results on a small set of test data. Therefore, a third generation of compiler design is proposed. These compilers analyze program behavior and report back to the user information concerning the efficiency and structure of the program. Using this information, the programmer should be able to modify the program accordingly.

This report contains several suggestions as to the type of data that can easily be generated by a compiler and be fed back to the user in order to accomplish this goal. The development of a data entropy measure will be described and the inclusion of several of these techniques into a diagnostic PL/1 system implemented at the University of Maryland will also be mentioned.

2. FLOW ANALYSIS MEASURES

It has been argued [4] that languages should not include any GOTO statement; however, the simple lack of a GOTO does not automatically lead to a well structured program since data also plays a significant role in program structure. An implementation that uses certain variables in every subroutine is not as well structured as one that localizes all accesses to only a few routines. This can be demonstrated by considering the problems associated with changing these variables. In the first case every subroutine must be studied and altered, while in the latter, only the few routines that actually use these variables must be changed. Parnas [14], among others, has been developing rules that allow for structured data.

Thus designing well structured programs consists of more than simply omitting all GOTO statements. The interesting question, therefore, is "What is meant by a well structured program?" Can this concept of structure be measured? From a pragmatic point of view, is this measurement effective, i. e. can a compiler be implemented to provide this information at minimal cost?

Several proposals have been made for providing some of this data. This section of the paper describes program traces as a measurement technique that gives a pictorial representation of some aspect of a program's execution.

2.1. Execution Profiles

One of the oldest data collection aids to be described is the execution profile [9]. A compiler can easily be altered to generate code to increment a count for each statement executed during program execution. This data can be used to produce a histogram or execution profile which graphically displays how many times each statement has been executed (fig. 1). Some of the advantages of such a system have been described by Ingalls [9]. Basically, the reasons for such a technique are:

1. Typically most of the execution time of a program is spent within a small section of the program; thus the execution profile will allow the programmer to optimize, by hand, those small sections of code that are frequently executed.

2. In a test debugging run, if any statement counts are zero, then the test data did not properly reflect actual program conditions since some program logic was either not exercised, or was faulty.

3. Execution profiles may also demonstrate unexpected properties about a program. It may turn out that a certain THEN clause may unexpectedly be executed more often than its corresponding ELSE clause. This type of data can be fed back into the compiler in order to better optimize the source program (as in the old FORTRAN II FREQUENCY statement).

In general the execution profile gives a condensed graphical picture of program execution. Due to the relatively short execution times for most debugging runs, the additional overhead in producing this data is well worthwhile. Depending upon the

```
EXECUTION HISTOGRAMS
EACH * =  65 EXECUTIONS

11       1
12       1
13      48
14      48
15      48
16       0
17      48
18      48
19       0
20      48
21      47
22      47
23      47
24       0
25      47
26      45
27      45
28      45
29      45
30      45
31      45
32      45
33      45
34     270     ****
35     225     ***
36     225     ***
37      45
38      47
39       1
40      45
41    1034     ***************
42     990     ***************
43       0
44     103     *
45     103     *
46     103     *
47     103     *
```

Fig. 1. Execution profile (partial listing). Note that some counts are zero since they correspond to nonexecutable statements like DECLARE statements.

type of data used, the execution profile can focus attention upon a small segment of the program that should be further studied by the programmer.

2.2. Static Language Analysis

Compilers can easily produce a count of each program's statements by type [11], and can easily generate code to keep track of how many times each statement type is executed (fig. 2). While this information is derivable from the source program listing and from the execution profile, the sheer volume of the data makes it almost mandatory that it be compiler generated.

This data can be used to discover general characteristics about a program. Relationships between static program structure (at compile time) and dynamic program structure (at execution time) may be studied. For example, the number of times that an IF statement is executed compared to the number of IF statements in a program may give an indication as to how well the input data is screened before it is used [10]. The collection of this data from many programs can lead to the development of general properties across many programs (fig. 4).

STATIC/DYNAMIC STATEMENT COUNTS

COMMENTS = 28 STATEMENTS = 85

EXECUTED STATEMENTS = 7521

TYPE	COMPILATION COUNT	%	EXECUTION COUNT	%
BEGIN	0	.0	0	.0
CALL	6	7.0	142	1.8
CLOSE	0	.0	0	.0
DECLARE	3	3.5	0	.0
END	15	17.6	1539	20.4
ENTRY	0	.0	0	.0
FORMAT	0	.0	0	.0
GET	1	1.1	48	.6
GOTO	0	.0	0	.0
IF	5	5.8	1175	15.6
OPEN	0	.0	0	.0
PROC	6	7.0	143	1.9
PUT	3	3.5	94	1.2
RETURN	0	.0	0	.0
STOP	0	.0	0	.0
NULL	0	.0	0	.0
DO	3	3.5	0	.0
DO WHILE	1	1.1	47	.6
DO ITER	5	5.8	1443	19.1
DO CASE	0	.0	0	.0
ASG GEN	11	12.9	1197	15.9
ASG 0 OP	2	2.3	48	.6
ASG 1 OP	24	28.2	1645	21.8

Fig. 2. Number and percentage of each statement type at compile and execution time (partial listing).

2.3. Trace History

A concept related to the execution profile is the concept of trace history. Let Kt be the set of statement numbers executed during time interval t. Kt is plotted vs. t to obtain a scatter plot of program execution vs. time (fig. 3). This data can easily be added to a compiler - especially to one that already has a statement tracing facility.

Using this data, the interrelationships among statements can be measured. It may show that certain statements are always executed in tandem with other statements.

```
STATEMENT NUMBER     EACH LINE REPRESENTS 100 STATEMENTS EXECUTED
         10        20        30        40        50        60        70
          +         +         +         +         +         +         +
-----------------------------------------------------------------------

  ***** *******
            ***
            ********* *           **     ***** ** *    *** **  ******
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                       *** ****           **********
                            ***
                 **** *     ****  **                   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   ******  **** *
                 **** ****        **     ***********   *** **  ** ***
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
                   ** **** ****          ***********   *** **  ** * *
                            ***
                 **         ****  **                   *** *
                 **** ****        **     ***********   *** **  ** * *
                 **** ****        **     ***********   *** **  ** * *
```

Fig. 3. Trace history. Vertical axis is execution time and horizontal axis is statement number.

For large programs it may show which routines should be grouped into single segments, and in a virtual machine environment it may give the programmer information on how to regroup subroutines in order to speed up execution time by reducing paging overhead. While the trace history has been used previously in the study of paging systems [12], it has not as yet been applied to study program behavior at the source language level.

2.4 Probabilistic Program Validation

It is also possible to view the execution profile as a probability distribution - the probability of being at a certain statement at any given time. With this approach, the same collected trace data can be used to compute a transition matrix giving the probability of transferring from one statement to another. If that is done, then some of the properties of Markov chain theory can be applied.

One possible application of transition probabilities is in a new definition of program validation. Since it is impossible to test a program for all possible input data, selected subsets of the data that are "representative" are chosen. One currently used definition simply states that a program is tested if every statement has been executed. As any programmer intuitively knows, this definition is extremely weak. An alternative definition has been that every path through the program has been executed. Unfortunately, the set of data needed to perform this testing is in general an undecidable problem. Therefore, the following definition is proposed: A data set tests a program if and only if the transition probabilities obtained are the same as the actual probabilities. Thus heavily used paths in the production program will be heavily tested in the test run, and lessor used paths will be less tested. Also, if the probabilities are the same, then most of the different execution paths will probably be tested. Note that this definition also includes the first definition of program validation mentioned above - if a statemet is ever executed, then its transition probability cannot be zero.

The problem with this definition is the determination of the actual transition probabilities. A proposed measure is to keep track of the range of values of the program variables. Using this range, dynamic programming techniques can be used to compute the probability of a conditional expression being true or false, and thereby giving estimates of the actual transition probabilities.

3. FEATURE MEASUREMENT

A related development to the above graphical techniques is the concept of feature measurement. This is the study of various numerical relationships that measure the overall quality of a program according to some norm or ideal. While these techniques do not give detailed breakdowns of the various attributes of a program, they do indicate trends in program structure. For example, as a teaching tool, if a student writes two versions of the same algorithm, and one has a different measure than the other, then one will be a better program according to some criteria. These measures, in conjunction with the graphical techniques already discussed, should lead to the feedback of information that should lead to a more well developed program.

3.1. Algorithm Dynamics

Halstead [6] has been studying algorithm size, and in the process has achieved some interesting results. One such result is that the approximation of program size is independent of programming language used. The number of tokens in the source program should be approximately:

$$a \log a \ + b \log b$$

where a and b are the number of distinct operators and operands in the program.

3.2. Program Work

Another trend is to compute the work performed by a program. Hellerman [7] has been studying the complexity of a function by computing the number of input variables that map to the same functional value. Let Xy be the number of input values that map to functional value y. The work performed by the function is then defined to be:

$$\sum_y Xy \log \frac{X}{Xy} = X \log X - \sum_y Xy \log Xy$$

where X is the total number of different input values.

In terms of measuring program efficiency, however, the program itself, and not the underlying function, should be measured. Data entropy is proposed as one such measure.

Van Emden [18] initially described a measure similar to the entropy of a physical system. Let {pi} be a partition of set P into sets of size $|pi|$. The entropy of the partition is defined as:

$$H = -\sum_i \frac{|pi|}{|P|} \log \frac{|pi|}{|P|} = \log |P| - \frac{1}{|P|} \sum_i |pi| \log |pi|$$

and is just the information content of a finite probability space.

If {A,B} is a partition of P, then the entropy loading of the partition is defined as:

$$C(A,B) = H(A) + H(B) - H(P)$$

Van Emden computed his entropy measure via an object/predicate table where:

Aij=1 if and only if object i had predicate j

For example, assume that the following set of 5 objects {1,2,3,4,5} has 5 predicates {a,b,c,d,e) as follows:

	a	b	c	d	e
1	0	1	0	1	0
2	1	0	1	0	0
3	1	0	1	1	0
4	0	1	0	1	1
5	0	0	0	1	0

In order to compute H({4,5}) consider only those columns containing information about either object 4 or object 5. The interrelationships among all objects, relative to these columns, will be measured. Object 4 is described by predicated b, d, and e. Object 5 is described by predicate d. Thus a reduced object predicate table can be prepared:

	b	d	e
1	1	1	0
2	0	0	0
3	0	1	0
4	1	1	1
5	0	1	0

From this data, the following partitions can be developed:

{1}, {2}, {3,5}, {4}

and the entropy can be computed:

$$H(\{4,5\}) = \log 5 - (2/5) \log 2$$

Similarly H({1,2,3}) and H({1,2,3,4,5}) can be computed, as well as the entropy loading of the partition:

$$H(\{1,2,3\}) = \log 5 - (2/5) \log 2$$
$$H(\{1,2,3,4,5) = \log 5$$
$$C(\{1,2,3\},\{4,5\}) = \log 5 - (4/5) \log 5$$

Van Emden has shown that for two different decompositions of P ({A,B} and {C,D}), if C(A,B) < C(C,D), then A and B interact less than do C and D; thus A and B are more independent.

Chanon [1] has used this measure in order to evaluate top-down programming. As a program is developed, assumptions are made about the data and an object/predicate table can be produced. Chanon showed that for two different decompositions of the same program, the one with the lower entropy loading was a more well structured version.

Unfortunately Chanon's idea cannot be used to automatically evaluate program structure via a compiler. Similar to the problems of automatically certifying the correctness of a program, the appropriate theorem proving techniques simply do not exist.

A modification to Chanon's approach, however, can be used to automatically generate structuring information. This new measure will be called data entropy of a program. Consider the attributes relevant to data storage: data may be known (declared) within a subroutine, data may be accessed, or data may be altered. Thus for each statement j (row in an object/predicate table) and for each variable i in a program let:

D(j,3i+1) = 1 iff i is known at j
D(j,3i+2) = 1 iff i is accessed by j
D(j,3i+3) = 1 iff i is altered by j

Using this definition, D forms an object/predicate table, and thus the entropy of a program can be computed.

This entropy measure has some of the properties desired of an entropy measure. It tries to measure the redundancy of data within a program - i. e. how many distinct variables actually represent the same physical construct. For example, in well designed systems, the data should be local to only a few routines. If that is so, then the entropy of the progam, relative to that data will be approxiamtely:

$$\log n - (k/n) \log k$$

where n is the number of subroutines and k is the number of routines that access the data. For small k the entropy will be maximal. (Note that this differs from the usual definitions of entropy where small values of the entropy measure mean less entropy. This conflict can easily be corrected by defining the measure as log n - H, since the maximal value is log n.)

4. IMPLEMENTATION

4.1 Implemented Measures

In order to test some of these ideas empirically, some of the previously described techniques have been implemented in a diagnostic system, called PLUM, implemented by the author at the University of Maryland.

PLUM is a load and go PL/1 compiler for the Univac 1100-series computer. It is typical of several compile and go systems in that it is based upon a very forgiving philosophy - most syntax errors are corrected automatically and most execution errors result in default values being used rather than having execution terminated. It is used primarily as a teaching tool, with the average student using under 5 seconds of computer time for each run [19].

The implementation of PLUM produces the execution profiles mentioned previously (fig. 1). Since the current statement number in execution was being saved in a register for diagnostic purposes, it was very easy to add code to update an array for each statement executed.

PLUM also produces a table giving the count of statement types in a program (fig. 2). Work is currently proceeding to modify the lexical scanner in order to have it generate the data necessary to produce the algorithm dynamics information.

The first implementation of the trace history (fig. 3) was a "quick and dirty" implementation that took about an hour to implement. Since PLUM already contains a tracing feature, the trace history was implemented by simply saving all traced output in file, and running a PL/1 program using this file as data. It will be a minor change to the runtime support routines in order to have the traced output save directly onto a mass storage file. In a similar manner, the transition matrix has been produced.

4.2 Development Tools

Aside from the measures that have been added to PLUM, additional features have been added which aid in developing well structured programs. This is especially important in a university environment where the compiler is a teaching tool in addition to being a program development aid.

A structured program is frequently a two dimensional program. Reading down the left margin gives the overall flow of the program, while reading to the right generally gives successively more and more detail as DO loops and IF statements are expanded. In order to facilitate this documentation process, an automatic formatter has been installed. Use of this option causes the source program listing to be indented for each nested DO loop or procedure block. This feature is convenient when statements are added as a program gets debugged and the source listing tends to get very messy (fig. 5).

Another feature which has been added is the printing of an error message for insufficiently commented programs. As of now, all programs must contain at least 10% comments or else a warning message will be printed. The next step will be to make this a terminal error message; however, before that can be done, more study must be done on the nature of program comments. Since this message has just recently been added, the reaction from the user community is eagerly awaited.

The ability to analyze many programs over long periods of time is important in evaluating the program development process. This ability has been added to PLUM via an automatic data collection facility. Each usage of the PLUM compiler causes approxi-

	ACCOUNTING			TIMES				PROGRAM SIZE				
ACCT #	ELT. NAME	OPTIONS	TYPE	COMPL	EXEC	# STMTS	OBJ	SYMTB.	INT FM	STACK	TOKS	BLKS
005223PAUL	ST1	ST	C+E	1116	2625	8	75	313	87	130	106	3
005223PAUL	ST1	CSTV	C+E	1150	2434	8	75	313	87	130	106	3
005223PAUL	ARINS	NT	C+E	1104	2399	4	45	191	36	115	40	2
005223PAUL	ARIN	NT	C+E	1076	2482	4	51	164	40	135	47	2
005223PAUL	ARINV	NT	C+E	1075	3098	11	103	232	73	151	81	4
005223PAUL	ENTF	NT	C+E	904	2334	9	26	192	51	87	57	2
005223PAUL	ENTR	NT	C+E	901	2332	9	25	192	47	87	55	2
005223K	DISPLAY	ET	C+E	984	2421	15	140	235	103	168	119	2
005223K	DISPLAY	ET	C+E	967	2476	15	140	235	103	168	119	2
005223K	DISPLAY	ET	C+E	993	22527	15	140	235	103	168	119	2
005223K	DISPLAY	ET	C	961	0	15	140	235	103	0	119	2
005223K	DISPLAY	ET	C+E	966	57226	15	140	235	103	168	119	2
005223MARV	P37	ST	C+E	527	2455	24	232	479	208	85	236	2
005223MARV	P3F		C+E	460	1501	[illegible]	99	345	134	178	153	2
005223MARV	P130	ST	C+E	457	2551	22	199	334	143	434	149	22
005223MARV	P171	ST	C+E	811	3044	44	702	649	480	970	807	2
005223MARV	P132	ST	C	534	0	29	373	512	279	0	318	2
005223MARV	F176	ST	C+E	448	1597	17	130	218	110	89	128	2
005223MARV	P177	ST	C+E	471	2711	24	205	316	175	399	172	2
005223MARV	F179	ST	C+E	491	2788	27	168	371	162	88	177	2
005223MARV	P140	STV	C+E	707	2239	44	306	723	335	322	364	2
005223MARV	FHARADA	EMNT	C+E	4535	6885	611	4508	4100	3436	1687	4173	749
005223MARV	P34	DST	C+E	555	8972	12	312	576	284	109	349	2
005223MARV	F30	DST	C+E	442	2120	19	108	335	124	89	124	2
005223MARV	PMAPS	ST	C+E	1274	4425	85	1089	910	786	1443	964	142

Fig. 4. Sample data collected on each program giving user account number, program name, number of statements, program size, and other characteristics.

```
ROTO:PROC OPTIONS(MAIN);
DECLARE (A,B,XMOLD,XM) FLOAT;
A=-2.;
B=-1.;
XMOLD=0.;
GOTO START;
GENER:DO;XMOLD=XM;
PUT SKIP LIST(XM);
START:XM=(A+B)/2.;
IF ABS(XM-XMOLD)<1.E-5 THEN GO TO FINISH;
IF F(XM)<0. THEN DO;
A=XM;
GOTO GENER;END;
ELSE  IF F(XM)>0. THEN DO;
B=XM;
GOTO GENER;END;
END;
F:PROC(X)RETURNS(FLOAT);
DECLARE X FLOAT;
RETURN(X**3-X+1.);
END F;
FINISH:END ROTO;
```

```
ROTO:PROC OPTIONS(MAIN);
    DECLARE (A,B,XMOLD,XM) FLOAT;
    A=-2.;
    B=-1.;
    XMOLD=0.;
    GOTO START;
    GENER:DO;XMOLD=XM;
        PUT SKIP LIST(XM);
        START:XM=(A+B)/2.;
        IF ABS(XM-XMOLD)<1.E-5 THEN GO TO FINISH;
        IF F(XM)<0. THEN DO;
            A=XM;
            GOTO GENER;END;
        ELSE  IF F(XM)>0. THEN DO;
            B=XM;
            GOTO GENER;END;
        END;
    F:PROC(X)RETURNS(FLOAT);
        DECLARE X FLOAT;
        RETURN(X**3-X+1.);
        END F;
    FINISH:END ROTO;
```

Fig. 5. The same program with the formatter turned off, and on.

mately 100 words of information to be saved in a mass storage file. Each entry consists of programmer name, program name, compile and execute time, and such program characteristics as number of statements, error messages and the static language analysis mentioned previously. (See fig. 4 for a partial listing of this data now being collected.) This automatic collection of data, unlike earlier semi-manual systems [13], will be used to answer questions such as: How does a single program develop as it gets debugged? What are characteristic errors in a program? And does the static language analysis undergo a similar evolution across a large class of programs as a program is developed?

5. CONCLUSIONS

There is currently no agreed upon quantitative definition of structured programming. It is not even clear as to what structured programs really are. Because of this, it is doubtful that automatic techniques can be developed in the near future to truly generate correct software.

However, a system can be implemented which does feed back information to the programmer which is of use in improving the structure of a program. Ideas are developing as to what information can be obtained, and how it can be used to produce better software. While a compiler may not know what reliable software is, it can let you know when you probably have achieved it.

ACKNOWLEDGEMENTS

The author is indebted to Paul McMullin and Keith Merkel for implementing many aspects of the PLUM system that have been described in this report.

REFERENCES

[1] Chanon R. On a measure of program structure. Symposium on Programming, Lecture Notes in Computer Science, Springer Verlag, 1974.

[2] Conway R., and W. L. Maxwell. CORC - The Cornell computing language. CACM 6, no. 6 (June, 1963) 317-321.

[3] Conway R. and T. R. Wilcox. Design and implementation of a diagnostic compiler for PL/1. CACM 16, no. 3 (March, 1973) 169-179.

[4] Dijkstra E. Goto statement considered harmful. CACM 11, no 3 (March, 1968) 147-148.

[5] Floyd R. Assigning meanings to programs. Symposium in Applied Mathematics 19 (1967) 19-32.

[6] Halstead M. and R. Bayer. Algorithm dynamics, ACM National Conference (1973) Atlanta.

[7] Hellerman L. A measure of computational work. IEEE Transactions on Computers C-21, no. 5 (May, 1972) 439-446.

[8] Hoare C. A. R. Hints on programming language design. Department of Computer Science, Stanford University, Technical Report 73-403 (1973).

[9] Ingalls D. The execution time profile as a programming tool. Compiler Optimization, Prentice Hall (1972), 107-128.

[10] Knuth D. An empirical study of FORTRAN programs. Software Practice and Experience 1, no. 2 (1971) 105-133.

[11] Lyon G. and R. Stillman. A FORTRAN analyzer. National Bureau of Standards Technical Note 849 (October, 1974).

[12] Morrison J. E. User program performance in virtual storage systems. IBM Systems Journal 12, no. 3 (1973) 216-237.

[13] Moulton P. G. and M. E. Muller. DITRAN - A compiler emphasizing diagnostics. CACM 10, no. 1 (January, 1967) 45-52.

[14] Parnas D. On the criteria to be used in decomposing systems into modules. CACM 15,no. 12 (December, 1972) 1053-1058.

[15] Ramamoorthy C. V. and S. B. F. Ho. Testing large software with automated software evaluation systems. IEEE Transactions on Software Engineering SE-1, no. 1 (March, 1975) 46-58.

[16] Rosen S., R. A. Spurgean, J. K. Donnelly. PUFFT - The Purdue University fast FORTRAN translator. CACM 8, no. 11 (November, 1965).

[17] Shantz P. et. al. WATFOR - The University of Waterloo FORTRAN IV compiler. CACM 10, no. 1 (January, 1967) 41-44.

[18] van Emden M. H. The hierarchical decomposition of complexity. Machine Intelligence 5 (1970), 361-380.

[19] Zelkowitz M. PLUM: The University of Maryland PL/1 system. Technical Report TR-318, Computer Science, University of Maryland, July, 1974.