An information structure for data base and device independent report generation

by C. DANA and L. PRESSER

*University of California**
Santa Barbara, California

INTRODUCTION

The generation of computer output information that is easily read by humans is a tedious and elaborate task when present-day programming languages are employed. This is certainly true when assembly language is used. Features like PL/I's *DATA* and *LIST* output options are improvements over FORTRAN's requirements of mandatory *FORMAT* statements. However, such desirable formats as tabular listings still require much programming and coordination. Another troublesome area involves the generation of information (i.e., reports) on devices that have different characteristics (e.g., printer, CRT). Typically, the size of the "page", and thus the amount of information that can be placed on a "page" will be different for the various devices. Current programming practice forces specification of the output format in a device dependent manner. Thus, to generate a report on more than one type of device would require recoding the section of a program that specifies the output format. Therefore, a method for describing just the logical format of a report, without consideration of the characteristics of the possible output device, would be desirable.

Once a device independent description of a report is obtained, it is a natural extension to attempt to separate the specification of the report from any given file or data base. Consequently, a generalized information structure for data base and device independent report generation is obtained. To generate reports it is necessary to implement a system that, with the device specification as a parameter, interacts with the informamation structure and the data base in order to generate actual output.

This paper describes a data base and device independent information structure for the representation of reports, the environment in which the structure resides, and the support programs that allow the generation of output.

It should be noted that a report need not be limited to business applications as is now generally thought. All areas of computer applications can benefit from uncomplicated output coding and from an orderly presentation of information. Indeed, there is a set of "output needs" that is common to most application areas. For example, observe the structural similarity between a report on employee's earnings (Figure 1) and a report on the performance of subroutines (Figure 2). It is our firm opinion that every (computer) system should incorporate at design time facilities for debugging and measurement purposes. Hence, the need to output information is inherent and ubiquitous, and thus, should be an integral consideration in the design of any programming language.

It is worthwhile at this point to discuss the general characteristics of a report. By a *report* we imply any visual computer output that is easily understood by humans. A report is not a static entity. That is, one can not, in general, lay down on a piece of paper the exact line and column positions of all items that will be part of a page and then complete the final report by just filling in the assigned areas with values. There can be sections of a report that may be repeated a number of times, the actual number being known at the time values are read from a data base. In Figure 1 each line in the body of the report corresponds to one employee, and the number of employees can vary from month to month. Similarly, in Figure 2, the number and frequency of subroutines used can vary from execution to execution. Therefore, since the final form of a report is not known until the data base is read, the logical description of a report must specify how input records are to be obtained, processed, and actual output generated. In general, one could conceive of a report whose final

* Department of Electrical Engineering. This work was supported in part by the National Science Foundation, Grant GJ-31949.

Page 1

EMPLOYEE EARNINGS

(A Sample Report)

Dept. #	Employee	Pay Rate	Hours	Total
351	John Smith	6.90	41	$ 282.90
	Ann Jones	6.50	45	292.50
	Peter Wilson	5.00	60	300.00
				$ 875.40
376	Mary Adams	5.50	55	$ 302.50
	James Peterson	6.60	44	290.40
	Henry Jennings	5.90	51	300.90
				$ 893.80

TOTAL PAID OUT = $1768.20

Figure 1—Employee earnings report

form would depend on a wide variety of relationships among variables. For instance, in our business example, we may wish to flag the names of those employees who have worked more than a fixed number of hours and whose pay rate is above a certain level. Similarly, in the measurements example, we may wish to flag the names of those subroutines that executed more than a fixed number of times and required more than a certain

Page 1

SUBROUTINE USAGE DATA

(A Sample Report)

Program	Subroutine	Number of Executions	Total CPU Time(ms)	Ave. time per execution
Program 1	A	3	129.51	43.17
	B	5	100.92	25.18
	C	1	71.32	71.32
Program 2	D	5	1100.31	220.06
	E	3	91.29	30.43
	C	1	39.98	39.98

TOTAL EXECUTION TIME = 1.532 seconds

Figure 2—Measurements report

period of time. Thus, a facility for testing relationships between variables and performing actions based on the results is needed for the logical description of reports.

The data base may not include all the values that are to appear in a report. For instance, we may wish to output total pay as part of a report when only pay rate and hours worked is present in the data base. Or we may desire to output average subroutine execution time when only total execution time and number of calls is present in the data base. Therefore, a facility to carry out calculations is needed for the logical description of reports.

In summary, the mechanisms needed for the logical specification of reports are the basic elements of a programming language. In fact, the information structure described here can be viewed as a special purpose report generating machine. It is also possible to view it as an intermediate representation for the translation of the output sections of programs. Indeed, it is this latter line of thought that motivated this work.

INFORMATION STRUCTURE ENVIRONMENT

Our environment for report generation is outlined in Figure 3. The user specifies, in some report generator

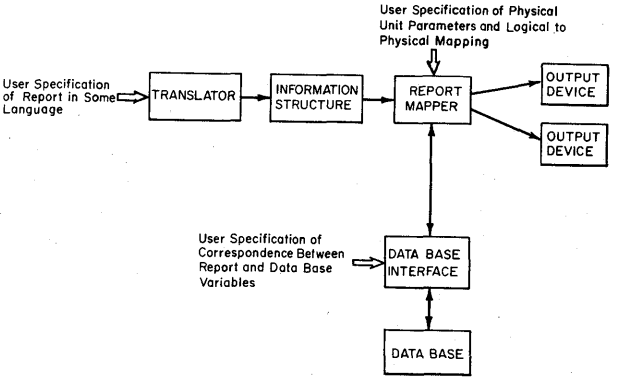
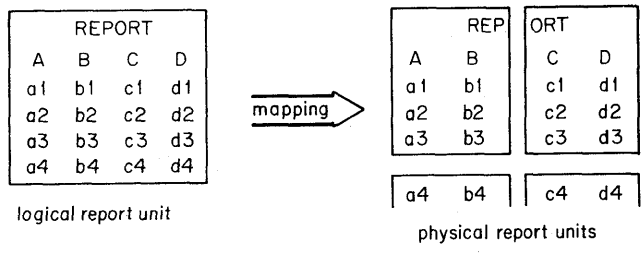
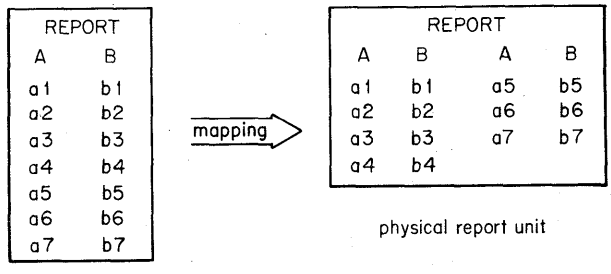


Figure 3—Information structure environment

language, the form of the desired report. For our purposes a *report generator language* is any language that possesses the facilities needed to describe the desired reporting. It may be a language designed specially for report generation (e.g., RPG), a more general language that includes special facilities for report generation (e.g., COBOL), or it may be a general purpose language (e.g., PL/I, assembly language). In the latter two cases the report generation code may only be part of a larger program.



a. Logical report unit larger than physical report unit.



b. Logical report unit narrower than physical report unit.

Figure 4—Sample mappings

The user's program is translated such that the logical description of the report is mapped into the information structure. (This information structure is discussed in detail in a later section of this paper.) The key information about the report, as originally described by the source language and now embodied in the information structure, is called the *logical report.* This is a specification of the report (e.g., where the headings are to be and what they are to state) in terms of some virtual (nominal size) surface called the *logical report unit.* The logical report unit consists of a fixed number of rows and columns. When the report finally appears on an output device media (e.g., paper, CRT face) it is called the *physical report* and the size of the actual display surface of the device is termed the *physical report unit.*

The mapping of a logical report unit into one or more physical report units is called the *logical to physical mapping* or simply the *report mapping.* Such a mapping includes obtaining data from the data base and the proper placing of results in the logical report, before generating a physical report unit. Report mapping is carried out by a program referred to as *report mapper* in Figure 3. Based on the information structure, the mapper fills out the logical report unit and then employs default or user specified parameters in order to carry out a logical to physical mapping. Examples of possible mappings are shown in Figure 4.

In general, soft-copy devices require mappings different from those employed with hard-copy units. In the case of a hard-copy device the logical report units can be split at arbitrary places to satisfy the physical report unit. The hard-copy segments can be later placed side by side and viewed as a whole. On the other hand, in the case of a soft-copy device the output can only be viewed one physical unit at a time; thus, care must be exercised to make each display coherent and readable. For example, the mapping shown in Figure 4a would not be very meaningful if the physical units must be viewed separately. In such a situation it is necessary to repeat identifying information and to make sure that all items are properly placed. Specific mapping algorithms are beyond the scope of this paper.

The report mapper buffers the physical report units before sending these units to the output device. The buffer size is important. If it is smaller than the logical report unit, placement of information would be restricted. For instance, a total could not be placed at the top of a physical report if the physical report unit that corresponded to the top of the physical report had already been sent to the device by the time the total was obtained.

In order to complete the discussion of the environment outlined in Figure 3 we need to describe the interface with the data base. The user must specify the correspondence between the variables present in the information structure and those residing in the data base. The function of the *data base interface* module (refer to Figure 3) is to supply the report mapper program with any data needed to generate a report.

INFORMATION STRUCTURE

The information structure consists of a number of tables (lists) each of which describes a section of the report format or generation process. Entries in each of the tables contain pointers to entries in other tables, thus, a linked structure is formed as depicted in Figure 5.

The *Report Head* describes the gross structure of the report. It contains the dimensions of the logical and physical report units. The latter may be supplied by the user or may be set, by default, to the value of the logical report unit. The report head also specifies any actions to be carried out at the beginning/end of the report and at the beginning/end of each logical page. The *report body action* entry is responsible for all of the other details of the report and, in essence, it represents the bulk of the reporting activity. The action entries in the *Report Head* point to a list of actions in the *Action Table.*

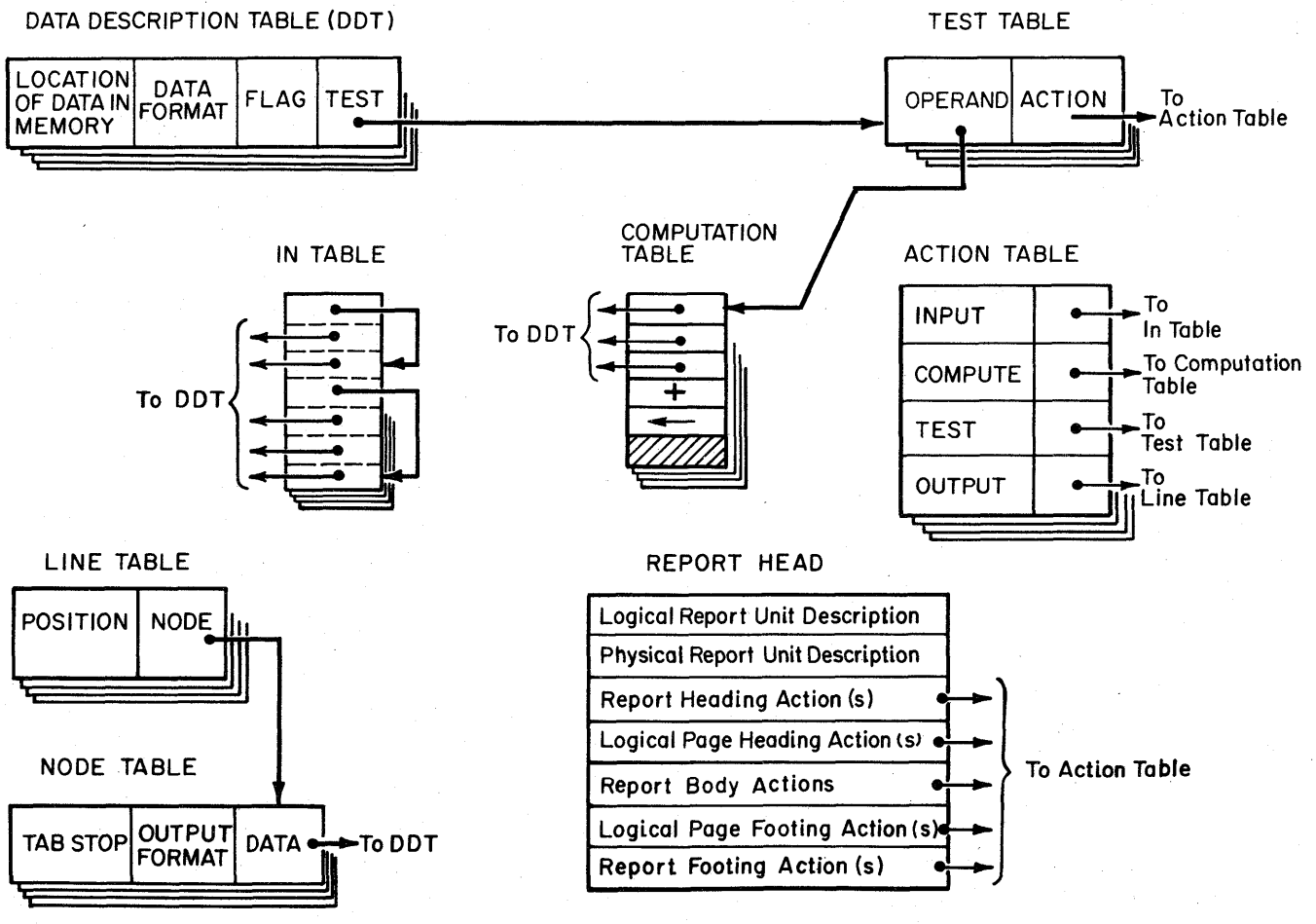


Figure 5—Information structure

The *Action Table* lists the sequence of actions that comprise the report generating process. There are four types of actions: *input* actions to obtain data from the data base; *compute* actions (including logical operations); *test* actions to determine flow of control; and output actions to create actual output. The detailed specification of these actions is contained in the *In Table*, *Computation Table*, *Test Table*, and *Line-Node Tables* respectively. Flow of control in the *Action Table* is sequential unless a transfer occurs as a result of a test.

The *Data Description Table* (*DDT*) contains information about the *location* and *format* of the data elements manipulated in the report. All references to data in any other table is specified by a pointer to a *DDT* entry. The *DDT* entry in turn contains a pointer to the location in memory where the actual datum is stored. There are two other fields in each *DDT* entry. The *flag* field is used to represent one of three possible conditions: (1) there are more data values to be input from the data base; (2) there is not more data available from the data base (i.e., "end of file"); (3) this datum represents an internal variable.* The *test* field specifies any test that is to be performed when the datum receives a new value. In essence, this facility implements an "on-condition". Such a capability may be exploited in the report generator language to free the user from having to specify a detailed ordering of calculations.

The *In Table* consists of a set of nodes, where each node is associated with an *input* action in the *Action Table*. A node consists of an ordered list of pointers to the *DDT*; the first entry of a node points to the last entry. The *DDT* pointers pinpoint which data values are to be input from the data base.

The *Computation Table* is a linear list. This table contains a postfix (Reverse Polish) representation of the computations to be performed on data. Each entry rep-

* Internal variables are those created to store intermediate values during report generation. It is assumed that a segment of memory is dedicated to auxiliary storage.

resents: an operand (i.e., pointer to the *DDT*), an operator, or an end of computation marker. The sequences of computations corresponding to *compute* actions in the *Action Table* are delimited by markers.

The *Test Table* contains the specification of the tests to be carried out and the action to be taken if the end result is true. The *operand* field points to the postfix representation of the test. The *action* field points to the action to be executed if the result is true. Note that tests are activated at two possible times: after manipulating (e.g., input operation) a *DDT* entry if the *test* field of the *DDT* entry is not null; or when control flows into a *test* in the *Action Table.*

The *Line Table* and the *Node Table* together specify the format of rows of the logical report described by the information structure. These tables support *output* actions. The *Line Table* defines line (row) position information for the lines of the logical report units. A line may have either a relative or absolute position specified in the *position* field. The relative position relates to the previous line and is employed with those sections of the report that may be repeated an indefinite number of times. For example, referring to Figure 1, relative positioning would be used to output a summary of employees' earnings when the number of employees to be reported is not known until the data base is read. An absolute line position corresponds to a fixed distance from the top of the logical report unit. It is used for those sections of the report whose positions will not vary from (logical report) unit to unit: for instance, page numbering. The segments of a line of output are defined by a list of nodes stored in the *Node Table.* The first node is specified by the *node* field in the *Line Table.* Each entry in the *Node Table* specifies the tab (column) position of the corresponding line segment, in the *tab stop* field; the external format in which data is to be output, in the *output format* field; and a pointer to a *DDT* entry for the data item to be output, in the *data field.* The *tab stop* field may also indicate a relative or absolute position.

Next, we discuss the support programs that allow the actual generation of output.

IMPLEMENTATION

The implementation of the system we have described is divided into two main parts. First, it is necessary to have a *Translator* that transforms a user's specification of a report into our information structure form. Such a transformation is not much different from that carried out by conventional translators; thus, it will not be discussed here. For an overview of the subject see Reference 1. The second part of the system is the *Report Mapper.* The basic design of this unit consists of a series of essentially independent subprograms. In terms of Figure 5, each subprogram relates to a particular table, and each performs the functions associated with the table in question. The subprograms employed in an experimental implementation are tabulated in Table I.

TABLE I—Subprograms in Report Mapper

Subprogram name	Associated Table	Function
ACTION	ACTION	Causes other subprograms to be called as directed by the Action Table.
COMPUTE	COMPUTATION	Performs the operation specified in the computation table.
DATA	DATA DESCRIPTION	Accesses information through DDT and passes it to other subprograms.
LINE	LINE	Causes a line of the report to be properly positioned and triggers beginning/ end of page actions.
NODE	NODE	Converts internal data format to external format and positions data in report buffer.
TEST	TEST	Causes execution of logical test and subsequent transfer of control.
INPUT	IN	Obtains new value for input variable(s).

As an illustration, let us examine how the *Node* subprogram operates in order to place a segment of a logical report line in a physical report. This subprogram is called by the *Line* subprogram and it is passed the address of the node describing the first segment of the line to be output. *Node* obtains a pointer to the *DDT* from the first entry in the *Node Table.* Next, with the aid of the *Data* subprogram, the desired datum is obtained as well as a description of its format. Then, the *data* and *output format* specifications are compared, and if they differ, a conversion to the otuput format specified by the *Node Table* entry is effected. Finally, the converted datum is positioned in a physical report buffer, as specified by the *tab stop* field of the *Node Table* entry and the report mapping algorithm. The remaining segments of the logical report line under consideration are processed in a similar fashion. The physical report is composed unit by unit in a buffer. When

the buffer is full it is output by a device dependent routine and reset to blanks.

It is worthwhile to observe that the implementation described here mechanizes a pseudo-machine whose instruction repertoire consists, in essence, of the four actions: *input, compute, test,* and *output.*

SUMMARY

In this paper we have presented an information structure for report generation that separates the data base and device dependent parts from the logical description of a report. Such a representation of reports (i.e., output) allows a clean and elegant interface to data bases and devices. It also brings out with some strength the need for better output facilities in current programming languages. Furthermore, it may serve as a guide in the design of report generator language facilities.

REFERENCE

1 L PRESSER
The translation of programming languages
In *Computer Science* A Cardenas L Presser and M Marin (Eds) John Wiley and Sons Inc New York 1972