DPL: A LANGUAGE FOR INSTRUCTION IN CONCEPTS BASIC TO DATA PROCESSING AND MANAGEMENT INFORMATION SYSTEMS*

Howard Lee Morgan

Technical Report
No. 68-15
25 April 1968

Department of Computer Science
Cornell University
Ithaca, N.Y. 14850

*Prepared for presentation at the Conference for Research on Management Information Systems, to be held at the Graduate School of Industrial Administration, Carnegie-Mellon University, 17-19 June 1968.

DPL: A LANGUAGE FOR INSTRUCTION IN CONCEPTS BASIC TO DATA PROCESSING AND MANAGEMENT INFORMATION SYSTEMS

Howard Lee Morgan[1]
Department of Operations Research
and
Department of Computer Science
Cornell University

1. Introduction

There is an even greater need for an instructional language for data processing than for scientific computing. More concepts must be covered and existing languages are in many respects unsuitable for introductory instruction. They obscure basic concepts with operational detail and lack facilities to illustrate contemporary systems design. The Data Processing Language (DPL) is designed to teach the basic concepts of data processing necessary to understand the design, programming, and operation of both conventional batch processing systems and of on-line inquiry and integrated management information systems.

The language does not compromise the goal of instructional effectiveness for the sake of generality or efficiency, and it is not intended to be used as a production language. The language processor includes extensive diagnostic aids, as well as error correction features. This type of language processor has proved its worth in teaching scientific programming [2], but has not yet been tried with a data processing language.

The concepts which are necessary in a contemporary

---

[1]The DPL design group consists of R. W. Conway, W. L. Maxwell, and the author. H. Krasnow and S. Saltzman participated in the early stages of the design. This work was supported in part by the National Science Foundation under Grant GP-6827.

data processing language include the following:

1. structured data; sequential and direct access files, records, arrays, tables
2. processing of text as well as numeric information
3. input and output; file and device oriented I/O
4. remote terminal handling; interrupts, polling
5. shared data bases; interactions between programs working on the same files.

These concepts are expressed in simple straightforward ways in DPL. Figure 1 on the next page provides a list of the DPL statements. The extra complexities and generalizations present in production data processing languages are left out of this language. In addition, the method used to illustrate shared data base concepts has many implications for the programming organization and design of management information systems.

The concept of structured data is the basis of data processing. Individual items of related data are grouped into records, and records containing information of a similar type are grouped into files. The manipulation of these files to produce desired summaries, reports, and other useful information is the aim of data processing programs. Section 2 describes the handling of these concepts in detail.

Some provision for handling remote I/O devices, such as teletypes, is needed for building an on-line inquiry system or a management information system. These devices typically demand the immediate attention of the central processor, regardless of any other concurrent processing. In addition to handling these interrupts, the system must also be able to regard these devices as the analogue of both a card reader and a line printer. Data processing programs also make heavy use of more standard I/O facilities to store and retrieve information from files, and to present information in a form

```
LET variable = expression
GO TO label
IF condition THEN statement ELSE statement
PERFROM block  expression TIMES
               WHILE condition
               FOR variable = e1 TO e2 BY e3
               WHEN condition
CANCEL block   WHEN condition
STOP
CREATE recordtype REF recref
DESTROY recref
OPEN filetype REF filref
CLOSE filref
READ recref [WITH key] FROM filref
WRITE recref IN filref
PROTECT filref
POSITION filref AHEAD expression RECORDS
                BACK  expression RECORDS
                AT HEADER
```

Figure 1. DPL Statements.

which may be used by managers. A consistent structure for handling all I/O has been adopted, and is described in Section 3.

In a management information system, there are usually many interactions between the information stored in the different files which make up the system's data base. An order entry, for example, may affect the customer file, open order file, inventory file, and the sales history file. In an integrated management information system, different programs may be used to maintain each of the major files. Also, this type of system should be able to handle transactions of many different types, e.g. payroll, order entry, etc. The basic block structure of DPL programs allows easy separation of the individual transaction processing routines and the file-tagging and program controlled interrupt features which are described in Sections 4 and 5 a-low many of these blocks to be interconnected in a flexible manner.

COBOL, the most widely used data processing language, has several drawbacks when it comes to teaching ideas and concepts. First of all, the verbosity of the language, which was intended to make programs self documenting, interferes with understanding which parts of the program are important and which parts are not. Second, the great flexibility in file, record, editing, field, and access method specifications cause the beginner to be caught up in a morass of detail without having seen the overall structure he is working with. Third, a contemporary language should be able to handle remote terminals and MIS design. While the random access and asynchronous processing additions to basic COBOL [6] are a step in this direction, the increased capabilities are gained through an even greater increase in complexity.

The following comments echo the author's feelings.

> Our School of Business [University of Wisconsin] tried teaching COBOL to beginning students, but dropped it. . . COBOL is much more useful at advanced levels, but their experience shows it takes too long to teach beginners to write useful COBOL.[2]

In summary, while COBOL allows the programmer to clearly see the trees (and even the leaves on the trees) it does not provide a good vantage point from which to observe the forest.

DPL, by eliminating verbosity, and severely constraining the types of program and data structures allowed, focuses attention on the concepts and fundamentals. With an understanding of these concepts, the student can progress to COBOL with a minimum of effort or difficulty. Again, this has been well demonstrated by the ease with which people who have been taught an instructional scientific language [2] progress to FORTRAN or ALGOL.

The implementation design is based on the Cornell University Programming Language, CUPL [4], [8], which is an instructional language for scientific computing. In order to enhance DPL's value as an instructional tool, the compiler and execution monitor are error correcting; whenever an error is detected, it is reported to the programmer, a reasonable correction is made, and processing continues. The diagnostic message given usually attempts to indicate the correct form of the statement in error, and will always tell the user what correction was made. The payoffs of this feature should not be underestimated. Studies of CORC programs [5] showed a

---

[2] These are comments by C. H. Davidson drawn from [7].

significant decrease in the number of passes needed to get a successful run. (A typical group of students required two passes in CORC as opposed to nine in FORTRAN to get a simple program running.)

DPL includes parts of CUPL, most notably the algebraic manipulation and logical testing statements. Certain features which are not often used in data processing (e.g., trigonometric functions) remain in the compiler for use in advanced applications such as corporate models. Other features of CUPL (e.g., the matrix algebra statements) are removed for economy of space.

It is intended that DPL be implemented for IBM 360 computers with at least 128K of core storage and one disc. The compiler will be written in a manner independent of the operating system, and will be reentrant and self-relocating so that it may be easily adapted to time-sharing environments. Although DPL includes the concept of sequential files, which have properties reflecting the characteristics of magnetic tape drives, the actual storage for all files will be on disc.

Appendix A provides a brief summary of the syntax and semantics of DPL programs.

2. Structured Data

Data Processing renders large amounts of information manageable by grouping it into hierarchical structures. DPL provides for the two levels of structured data most often found -- variables grouped into records, and records grouped into files. A third structure, which may exist within or outside of the above hierarchy, is also provided -- variables grouped into a subscripted array. Structures are defined at the beginning of the program in a Data Description section.

Files may be of two types, sequential or direct access. Both types must begin with a header record. Direct access files may contain only one type of record in addition to the header; these records are logically arranged in increasing order of a key variable. Sequential files can contain many different types of records in addition to the header. They should normally close with an end-of-file record, but this is entirely the programmer's responsibility and it is not treated differently from other records.

Simple variables and arrays which are not part of records must be listed at the beginning of the Data Description section, before the first file-type entry. The description of the structure of a particular file begins with the line on which the file-type is given, and continues to the line with the next file-type entry (or to the end of the Data Description section). The structure of a record continues to the line on which the next record-type entry begins.

A particular record-type may be used in more than one file-type. The listing of the elements of a record-type need only appear at the first occurrence of this entry. Subsequent references may, however, repeat one element in order to flag this element as a key variable for a direct access file. Except for this multiple use of a particular record-type, all identifiers in the Data Description section must be unique. This means that a particular variable name may occur in one and only one record-type, and hence there is no analogue to COBOL's MOVE CORRESPONDING statement.

A record is unlike a simple variable in the following important respect. When a variable is declared, its "name" is assigned and storage space is allocated to contain its value. This name is used to represent its current value in expressions, etc. When a record-type is described, a description of the structure of the record is composed, but <u>neither a name nor storage space are assigned at the time of declaration</u>. Both of these assignments are accomplished during execution of the program through the CREATE statement.

For example, the record-type PAYREC is declared in the sample Data Description (Figure 2) to consist of three variables (CLOCKNUM, NAME, CUMEARN). During execution one might encounter the statement:

CREATE PAYREC REF P1

At this time four words of storage are set aside to be a PAYREC and the name P1 is assigned to this particular PAYREC. If one wishes to refer to this particular record for processing, he does so by name -- P1, and not by record-type. In general, there will be several similar CREATE statements, each creating a PAYREC, but each assigning a different name to the particular PAYREC. These names have no value that is intelligible to the user and are never displayed. They are used as pointers to the storage space assigned to the particular record. They may, however, be used in assignment statements just like all other variables. For example, the statement:

LET RECNUM = P1

would create a pointer to the same record P1 points to, in a variable named RECNUM.

DPL DATA DESCRIPTION

FILE TYPES NAME	(a)	(b)	RECORD TYPES NAME	KEY	VARIABLES NAME	SUBSCRIPT BOUNDS 1ST	SUBSCRIPT BOUNDS 2ND	FORM
					P1			R
					J			N
					RECNUM			R
					LIST	30		R
					OLDMASTR			R
					NEWMASTR			R
PAYFILE	SQ	H	HEADER		TYPE			T
					DATE	2		T
			PAYREC		CLOCKNUM			T
					NAME	2		T
					CUMEARN			N
			OVERTIME		AMOUNT			N
[illegible]	DA	H	HEADER					
			ACCOUNT	K	ACCTNUM			N
					AMT			N

(a) Sequential (SQ) or Direct Access (DA).
(b) 'H' for one record type per file. Indicates header record type.
(c) 'K' for one variable for each DA file. specifies the key variable.
(d) Specifies the form of the variable. N= number, T= text, R = reference. Default assumption is number.

Figure 2. Sample DPL Data Description

Similarly, the name of a file is assigned by the OPEN statement during execution and not by the declaration in the Data Description. There may be several files of the same type open at the same time. For example,

```
OPEN PAYFILE REF OLDMASTR
OPEN PAYFILE REF NEWMASTR
```

would create two files of type PAYFILE with different names.

Variables in DPL contain information in one of three forms, regardless of whether the variable is simple, array, or part of a record. The <u>number</u> form is the same as the form of all variables in CUPL -- a double word (64 bit) floating point number. Automatic relative roundoff routines are used when integers are needed (e.g., as subscripts or loop counts) so that there are no mixed mode conventions to learn. The <u>text</u> form contains eight characters of alphanumeric information. The <u>reference</u> form is used for variables that will be used as file, record, or device names. Their value is a pointer intelligible only to the system, and not to the user. The form of a variable is specified in the Data Description section and remains in force for the life of the program. The number form is assumed if no specification is given.

A simple variable is referred to by its name. An array variable is referred to by its name followed by one or two subscripts enclosed in parentheses, as appropriate. A variable which is part of a record is referred to by a compound identifier consisting of the record name, a period, and the element name: P1.CLOCKNUM would refer to the second element of a PAYREC named P1.

Text constants anywhere in the program are enclosed in quotes:

```
LET P1.NAME(1) = 'JONES,A'
IF P1.CLOCKNUM NE 'J0245" THEN GO TO UPDATE
```

Text variables can be manipulated using the algebraic and logical statements, but no real facilities for string or single character manipulation (e.g., substring matching) have been provided. Strings longer than eight characters can be stored in text arrays, and will be printed out as a single continuous string.

3. Input and Output

Data Processing programs usually have heavy I/O requirements and any language which attempts to teach data processing must pay special attention to I/O. For instructional purposes it is convenient to divide I/O operations into two categories -- file-directed and device-directed. File-directed I/O operations are those which involve a transfer of a record to or from a file. The position of the record in the file is determined by the logical structure of the file. Also, a file-directed operation usually has both record and file in machine usable form. A device-directed operation is one which involves a transfer of information to or from a specified device. There need be no logical connection between the information transferred to or from the device on successive transfers. A conversion from machine usable to man usable form, or vice versa, usually occurs in a device-directed operation. Conventional batch processing systems are mainly file-directed, while on-line inquiry or management information systems are more evenly divided between file-directed and device-directed operations.

An example may help to make this division clearer. Suppose a summary of employee earnings is being printed. First, a page heading would be output to the line printer (device-directed). Then, an employee record would be read from the employee file (file-directed), and selected information from this record would be output to the printer. Records would continue to be read and printed until, say, 55 had been printed. A page eject command might then be output to the printer. The logical structure of the file specifies the next record input on a read, while the next record to the printer is specified by a characteristic of the device, i.e. 55 lines to the page are desired.

Naturally, a file must reside on some physical device (a disc in DPL), but the concept of a file implies a logical group of data rather than the physical device on which that data resides. Some modern operating systems recognize this and partially remove from the programmer's consideration the precise physical location of their files [1].

Every I/O statement in DPL specifies some pieces of information (variables, records, or arrays), and the file or device to which (or from which) the information is directed. Except for two special devices -- the system input unit (card reader) and the system output unit (high speed printer) -- the file or device must be explicitly referred to by the name given it in an OPEN statement.

If devices other than the reader and printer are desired, they may be described to the system by a device declaration statement. An example will illustrate this specification.

```
DEVICE: TELETYPE,MAX=4,INPUT,OUTPUT,INTERRUPTS
```

This would tell the system that there are up to four devices of device-type TELETYPE available to the system, each of which is capable of both input and output, and each of which can generate interrupts. All device declarations must appear before the first executable statement of the program and after the Data Description section. Before issuing commands to a device, the device must be given a name in the same way that a file is named, i.e., through an OPEN statement:

```
OPEN TELETYPE REF LIST(J)
```

In the case of devices which can generate interrupts, the OPEN has the effect of enabling the interrupts.

When a user is finished using a particular file or device, he issues the statement:

```
CLOSE LIST(J)
```

to close the file or device referred to by LIST(J). If a user has written 100 records on a PAYFILE named LIST(J), and then closed that file, he may read the records by issuing another OPEN statement of the form:

OPEN PAYFILE REF LIST(J)

at any time before he uses the variable LIST(J) to refer to something else. If the user wishes to save a file across job boundaries so that another user can reference it, he issues the statement:

CLOSE LIST(J) SAVE dsname

where dsname is a legal dataset name defined by the conventions of the installation. Another user can then access this file in his own program by writing:

OPEN PAYFILE REF P1 USING dsname

Of course, this user must have the same file description for PAYFILE as did the person who created the file.

File-directed I/O operations involve reading or writing a specified record of a file. For sequential files, the record is specified by a position pointer associated with the file. Each time a record is read or written, the pointer is advanced to the next record. This corresponds to the physical movement of a magnetic tape, but the logical sequence of records is the concept here. For direct access files, the value of the key element is used to specify which record is to be accessed. Direct access files have their records arranged in ascending order of the key variable. Hence, a READ statement which does not specify a value for the key will advance the pointer to the next record, i.e. the record with the next higher key. This allows a direct access file to be read sequentially even though it was not written sequentially.

The user may reposition the pointer associated with a file at any time by executing a POSITION statement, e.g.

POSITION P1 AHEAD 30 RECORDS

To read or write records from the files described in the sample Data Description section (Figure 2.), the following statements might be used:

```
READ RECNUM FROM OLDMASTR
READ LREC WITH ACCNO FROM LIST(12)
WRITE P1 ON LIST(J)
```

In the second statement above, the variable ACCNO is assumed to contain the value of the key of the account record we are interested in, and LIST(12) refers to a file of type LABDISTN.

Often, a user will wish to insure the safety of a particular file from being destroyed or overwritten. The statement:

```
PROTECT OLDMASTR
```

will make the specified file read-only until the file is closed. Any attempt to write in such a file will not be executed and an error will be generated.

The programmer is informed of end-of-file or no-record-with-specified-key conditions through a special function, TEST. If a READ statement cannot be executed for one of the above reasons, a failure flag is set, and can be tested by means of the statement:

```
IF TEST(LAB1) = 'FAIL'  THEN GO TO NOREC  .
```

The failure flag will be set for other types of I/O errors, but in cases other than the two mentioned above, standard error messages and corrections will be generated.

Device-directed I/O operations may specify lists of variables, records, or text information and the device to or from which the transfer is directed. If no device is specified in a READ statement, the card reader is assumed. Similarly, the printer is the default for a WRITE command. When a record is named in a WRITE statement, the record-type will be output as well as each of the elements. On input, the record name must be present as the first element of data from the device.

Output lines are made up of six fields; each of which may contain either a number, text information, or a variable name. When a WRITE is given, successive elements are assigned to successive fields on the output line. There is a special provision for printing amounts of money. If a variable name in a WRITE statement is preceded by a $, the name of the variable will not be printed, a $ will be printed immediately to the left of the first significant digit, and only two places will be printed after the decimal point. Appendix A contains a more detailed description of the actions which are allowed in a WRITE statement.

4. Interrupts

A management information system may be divided into four parts. First, there is a data base -- all of the information which is available to the rest of the system. Second, there are programs used to keep the information in the data base current, e.g., data entry or updating programs. The third division includes those programs which utilize the data base in a read-only manner, such as on-line inquiry systems. Finally, there is a supervisory program which schedules the execution of the other programs on a "when needed" basis. It is intended that DPL be a language in which one can describe and program all four of these parts.

The structural data concepts described in Section 2. allow one to set up the data base as a collection of files. Programs in the second and third groups above may be written using the usual algebraic and logical manipulation statements and the input output abilities discussed in the previous section. It is to the specialized concepts needed to program a supervisor that the rest of this section is directed.

The supervisor must know what conditions require the execution of each of the other programs in the system, and must have a means of determining when these conditions have occurred. In DPL, interrupts are used to indicate the occurrence of these conditions, and the two types of interrupts felt to be most useful in constructing a management information or on-line inquiry system are allowed, namely, device interrupts (such as from a remote terminal), and program controlled interrupts. In both cases, the interrupt is processed by an interrupt function module[3] which is written as a DPL block.

---

[3] See [9] for a discussion of the interrupt function module concept.

(A block consists of a sequence of statements delimited by the two statements:

```
blabel BLOCK
blabel END
```

It can be executed only by executing the DPL statement:

```
PERFORM blabel [various modifiers]
```

where blabel is an identifier. The modifiers are described in Appendix A. The block concept is used to teach both iteration and subroutine, although there is no transfer of parameters and all variables in a block are global.)

The programmer indicates which blocks process device interrupts by using statements of the form:

```
PERFORM TTINT WHEN LIST(4) INTERRUPTS
```

where LIST(4) refers to a device which can generate interrupts, and TTINT is a block as described above. If an interrupt is received from the device, execution of the currently executing statement is completed, the block TTINT is performed, and control returns to the statement which would have been next. Nested device interrupts are not permitted, i.e., if LIST(4) generates an interrupt while TTINT is being executed, the new interrupt will no be handled until TTINT is completed.

The real power of the system, however, and the feature which has the most implications for management information systems programming, is the program controlled interrupt feature. Programs may specify conditions under which a block should be performed by using any Boolean expression after the WHEN clause in a PERFORM statement. For example,

```
PERFORM REORD WHEN STKLEV LE REORDPT
```

would tell the system to schedule the execution of the block, REORD, whenever the value of STKLEV is less than or equal to the value of the variable REORDPT.

When the above PERFORM statement is executed, the block name and the associated condition (denoted as the pair (b, c)) is placed on the "pending block" list, and all variables used in the condition have a flag set in the main symbol table. This essentially tells the system to generate an interrupt whenever the condition is true, and to use the block named as the interrupt function module. On any subsequent store into a variable which has been flagged, all conditions which involve that variable are checked. If any are true, the (b, c) pair is placed on the "to be executed" list and removed from the "pending block" list.

Note that the condition is not checked when (b, c) is added to the "pending block" list, but rather on subsequent stores. This permits the last statement in a block to be a PERFORM WHEN which reschedules(b, c) .. A pair can be removed from the "pending block" list by executing the statement:

CANCEL blabel WHEN Boolean condition

where (blabel, Boolean condition) is identical to a pair on the list.

The situation can become complicated if there are several blocks on the "to be executed" list as the result of the change in value of a single variable, or as the result of executing some of the blocks from this list. Figure 3. on the next page presents the algorithm used to schedule the execution of blocks from the "to be executed" list.

An example might help to explain the reasoning used in constructing the algorithm. Suppose one block is used to print an exception report whenever the variable X takes on the value 2, while another block is supposed to change the value of X to 3 whenever it becomes 2. Presumably, there is a reason for these blocks to be written separately. Let the blocks be named B1 and B2 respectively. Say we execute

ALGORITHM A.

1. Label all (block, condition) pairs on the "to be executed" list (tbe) as level 1.

2. Select from the "tbe" list all blocks whose execution will not change any of the conditions which are on the "tbe" list (i.e. those which make only read-only demands on variables which are in conditions on the list). Execute these blocks and remove them from the list.

   Note a: If any new interrupts are generated by the execution of any of these blocks, add the new (b, c) pairs to the "tbe" list marked as level 2, and remove these pairs from the "pending block" list.

   Note b: If PERFORM . . . WHEN statements are issued by any of the blocks being executed, add the (b, c) pairs to the "reschedule" list, and not to the "pending block" list.

3. For the (b, c) pair next on the "tbe" list with level marked as 1, test the condition, c. If c is true, go to step 4. If c is false, go to step 3. If we have reached the end of the "tbe" list, go to step 5.

4. Execute the selected block and remove the associated (b, c) pair from the "tbe" list. Notes a and b above apply to this execution. When execution is completed, go to step 3.

5. Flag all blocks remaining on the "tbe" list at level 1 as "in conflict", and remove these pairs from the list. (see Section 5 for a discussion of "in conflict".) If the list is now empty, go to step 6. If not, go to step 1.

6. Add the (b,c) pairs on the "reschedule" list to the "pending block" list and delete the "reschedule" list. Return control to the main program at the point where the interrupt occurred.

Figure 3. Algorithm A.

the statements:

```
PERFORM B2 WHEN X = 2          and
PERFORM B1 WHEN X = 2          .
```

Since (B1, X=2) involves a block which has read-only access to the variables involved in the condition, it is performed first, and the report is printed. Then, since the condition associated with (B2, X=2) is still true, B2 is executed and changes the value of X to 3. If the blocks had issued any PERFORM WHEN statements, these (b, c) pairs would then be added to the "pending block" list.

Further study of conflicts of the type which occur above is certainly necessary, but it is important not to confuse this problem with the problem of multiple users wishing access to a file or other common resource, which can be solved in a fairly straightforward manner by a technique similar to that presented in [3].

--

5. Shared Data Bases

Section 3. described how files could be saved and made available to other users of the system, and Section 4. developed the program controlled interrupt features. In this section the two concepts are combined to simplify the large effort which usually goes into the organization and programming of an "integrated" management information system.

Some of the variables which are part of the conditions on the "pending block" list may be elements of files. For example, the user may have specified:

```
PERFORM REPORT2 WHEN RECNUM.CUMEARN GE 10000 .
```

If a file being used by some program is closed with the save option while there are still conditions involving elements of that file on the "pending block" list, a record of these conditions, and the associated interrupt function modules is stored with the file. Then, when any other user opens that file for processing by his program, the system adds any stored conditions to the pending list, just as if the current user had issued them. The first user has, so to speak, put some tags on the file to indicate his interactions with the elements of that file. This feature is therefore called "file tagging."

A management information system can thus be composed of a large group of interrupt function modules, together with a collection of tagged files. Each programmer indicates to the system which conditions he is interested in, and which routine should be used for each condition. The method is similar to that of decision tables, but with asynchronous checking of the lines of the table, although this is rather oversimplifying the issue.

Algorithm A contains a conflict recording mechanism. The system will use this mechanism to inform a particular programmer that one of his blocks is in conflict with some other block for certain elements of a file, and the two can then resolve their conflict manually. It is hoped that this will be more efficient than the usual practice of trying to make one person responsible for a file, even though there are many other programmers who have an interest in some of the elements of that file.

6. Summary and Conclusions

DPL is a language for instruction in data processing, and as such should not be expected to perform tasks in the most efficient manner, or in manners directly analogous to production languages. It also contains some new ideas in the area of management of shared data bases, which will be explored further in settings other than the instructional one.

It is hoped that implementation will be completed by early summer of 1968, and that use of the system can begin in the fall of 1968. Reports on the ease of teaching data processing, the usefulness in exacting an understanding of management information systems, and the ease of progressing to a production data processing language will be made as soon as sufficient data is gathered.

At this early stage in DPL's development, any comments from interested readers or prospective users would be welcomed.

REFERENCES

1. Bash, J. L., et. al. The MULTICS Operating System, Cambridge Information Systems Laboratory, May 1967.

2. Conway, R. W. and W. L. Maxwell, "CUPL - An Approach to Introductory Computing Instruction," Technical Report No. 68-4, Department of Computer Science, Cornell University, Ithaca, N.Y., 1968.

3. Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control," CACM 8, (September 1965).

4. Elder, H. A., "CUPL - System's Programmers Information Guide," (unpublished). Available from the Office of Computer Services, Cornell University, Ithaca, N.Y.

5. Freeman, D. N., "Error Correction in CORC," Proceedings of the AFIPS Fall Joint Computer Conference, 1963, Spartan Books, Washington, D. C., 1963.

6. IBM 360 Operating System - COBOL Language, C-28-6516-6, IBM Data Processing Division, White Plains, N.Y., 1965.

7. Organick, E., "Panel Discussion on Computer Appreciation," CACM 11, (April 1968).

8. Walker, R. J., CUPL - The Cornell University Programming Language, Department of Computer Science, Cornell University, Ithaca, N.Y., 1966.

9. Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill Book Company, N.Y. 1968.

Appendix A

Summary of DPL

ELEMENTS OF THE LANGUAGE

Characters

Letters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits: 0 1 2 3 4 5 6 7 8 9
Special Characters: + - * / ( ) . , = $;

Numbers

Normal decimal usage: e.g., 3, 1.725, -.06
"Scientific" notation: -1.2E-42
Truncated to 9 significant figures on input.
Range: Absolute values from 10^{-78} to 10^{76}, and 0.

Identifiers

a. Consist of 1 to 8 letters or digits, beginning with a letter; no blanks or special characters allowed.
b. Must not be one of the following "reserved" words:

ABS	CLOSE	FOR	LET	PERFORM	SAVE	WHEN
AHEAD	COMMENT	FROM	LN	POSITION	SIN	WHILE
ALL	COS	GE	LOG	POSMAX	SQRT	WITH
AND	CREATE	GO	LT	POSMIN	STOP	WRITE
AT	DESTROY	GT	MAX	PROTECT	TEST	
ATAN	DA	HEADER	MIN	RAND	THEN	
BACK	ELSE	IF	NE	READ	TIMES	
BLOCK	END	IN	ON	RECORDS	TO	
BY	EXP	INTERRUPTS	OPEN	REF	TYPE	
CANCEL	FLOOR	LE	OR	SQ	USING	

c. Must be unique; the same identifier cannot be used for more than one type (i.e., variable and label).

Files, Records, Devices, Variables, and Labels

a. Names for file, record, and device types, and variables and labels, are all identifiers as described above.

b. All of the above except for labels must be declared in the Data Description section.

c. Record elements and variable names can be singly or doubly subscripted.

d. Variables can store one of three forms of information; number (64 bit floating point), text (8 alpha characters), or reference (pointer to record, file, or device). This form is declared in the Data Description section and is fixed for the life of the program.

e. Simple variables are referred to by name. Subscripted variables by the name followed by one or two expressions enclosed in parentheses and separated by commas, e.g. A(I,J+4*K). Elements of a record are referred to by record reference variable, a period, and the element name, all subscripted as necessary, e.g. LIST(J).NAME(2).

f. Labels may be singly subscripted, with up to 10 diffe-ent subscript values, each of which can be either numeric or alphabetic, e.g., PROCESS('ORDENT') .

<u>Arithmetic Operators</u>

a. +, -, /, * for multiplication, ** for exponentiation.

b. Precedence: **, * and /, + and -. Parentheses from inner to outer. Sequence of equal precedence from left to right.

<u>Spacing</u>

No spaces, or splitting at end of line, in any number, variable, label, reserved word, or **, or textstring. Spaces are allowable anywhere else.

<u>Functions</u>

a. Arithmetic arguments: ABS, ATAN, COS, EXP, FLOOR, LOG, SQRT, SIN, MAX, MIN, RAND.

b. Vector arguments: POSMAX (gives row position of maximum element in a vector), POSMIN (same for MIN).

c. File Reference arguments: TEST (gives 'FAIL' if last I/O reference to that file failed), TYPE (gives eight character string of type of next record on the file or device pointed to).

<u>Relations</u>

a. =, NE, LE, GE, LT, GT .

b. All can be used with both numeric and alpha data. All numbers are NE to all alphanumeric variables.

Statements

The following symbols are used in the statement descriptions:

$v_1, v_2, \ldots$ variables (text or numeric)

$fr_1, fr_2, \ldots$ reference variables pointing to files

$rr_1, rr_2, \ldots$ reference variables pointing to records

$c_1, c_2, \ldots$ relational operators

$slabel_1, slabel_2 \ldots$ statement labels

$blabel_1, blabel_2 \ldots$ block labels

$e_1, e_2, \ldots$ arithmetic expressions (a meaningful combination of numbers, variables, record elements, functions, and arithmetic operators)

Any statement may be given a label -- beginning in column 1 of the programming form. Statements should begin in column 10 of the form. If continued onto more than one line, the second and subsequent lines should begin in cloumn 15. Columns 73 to 80 must not be used.

Assignment Statement

LET $v_1 = e_1$ The value of the expression is assigned to the variable on the lefthand side. IF v_1 is a text variable, e_1 must be either a text constant or a single text variable.

Sequence Control Statements

GO TO $slabel_1$

GO TO $blabel_1$ END Used only inside block '$blabel_1$', causes skip to end of block.

IF e_1 c_1 e_2 THEN s_1 ELSE s_2 where s_1 and s_2 are any type of statement except IF. Either the THEN or ELSE phrase may be omitted, but one or both must be given.

Compound conditions may be used:

e_1 c_1 e_2 AND/OR e_3 c_2 e_4 AND/OR ...

but AND and OR phrases may not be mixed in the same statement.

STOP

Iteration Control Statements

A 'block' consists of a sequence of statements preceded by

blabel$_1$ BLOCK and followed by blabel$_1$ END.

A block may be located anywhere in the program; it is executed only by a PERFORM statement calling it by name. Blocks may be nested but not overlapped. A block may contain any kind of statement, including PERFORM, except for a PERFORM which refers to the block itself for immediate execution (i.e. all except PERFORM WHEN).

PERFORM blabel$_1$ — Executes the block once.

PERFORM blabel$_1$ e_1 TIMES where e_1 has integer value.

PERFORM blabel$_1$ WHILE e_1 c_1 e_2 — Executes the block while the condition is true. Compound conditions may be used with the same restricitons as IF.

PERFORM blabel$_1$ FOR v_1 = e_1, e_2, e_3 ...
FOR v_1 = e_1 TO e_2 BY e_3 — The BY phrase can be omitted if e_3 =1.

PERFORM blabel$_1$ WHEN e_1 c_1 e_2 — The pair (blabel$_1$, $e_1c_1e_2$) is placed on the pending block list. When the condition becomes true, the block will be executed once.

CANCEL blabel$_1$ WHEN $e_1c_1e_2$ — The pair (blabel$_1$,$e_1c_1e_2$) is removed from the pending block list

Storage Management Statements

CREATE rectype REF rr$_1$ — This allocates space for a record of type rectype, and places the pointer in rr$_1$.

DESTROY rr$_1$ — Frees the storage associated with rr$_1$ and sets rr$_1$ to NULL.

Communication Statements

OPEN filtype REF fr_1
OPEN devtype REF fr_1 — Creates an instance of the file or device type specified, and points to it with fr_1. If fr_1 had previously pointed to a file of this type, the file is reopened.

CLOSE fr_1 — Closes the file or device.

PROTECT fr_1 — Causes the specified file to be made read-only. Any write references to it will be considered errors.

POSITION fr_1 AHEAD e_1 RECORDS
BACK e_1 RECORDS
AT HEADER — The pointer associated with the next record in the file is moved to point to a different record, as directed.

READ v_1, v_2, ... — Variables are read from the system input unit.

READ rr_1 FROM fr_1 — The next record from file fr_1 is read into rr_1

READ rr_1 WITH e_1 FROM fr_1 — The record from the Direct Access file fr_1 with key = e_1 is read into rr_1.

WRITE v_1, rr_1, 'message', /v_2, \$$v_3$

Five types of elements may appear on the list after a write directed to the printer:

1. v_1: prints name of variable and current value.
2. rr_1: prints record name and values of all elements.
3. message: prints the character string in quotes.
4. /v: prints the value of the variable only.
5. \$v: prints a \$ to the left of the most significant digit of the value, and two decimal places.

WRITE rr_1 IN fr_1 — Writes the sepcified record into file fr_1. Position in the file is determined by key if direct access, else record is added to the end of the file.

Comment Statement

The word COMMENT in columns 1-7 of the card, followed by any text desired will cause the card to be listed with the source program only.