# DPL: A Language for Instruction in Contemporary Data Processing Concepts

Howard L. Morgan

TECHNICAL REPORT
No. 68-24

September 1968

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

# DPL: A LANGUAGE FOR INSTRUCTION IN CONTEMPORARY DATA PROCESSING CONCEPTS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University for the Degree of

Doctor of Philosophy

by

Howard Lee Morgan

September 1968

## BIOGRAPHICAL SKETCH

Howard Lee Morgan was born in New York City on November 14, 1945. He attended elementary schools in the Bronx and graduated William Cullen Bryant High School in Queens, N. Y. with honors. In the fall of 1962 he entered the Selected Student Program at the College of Liberal Arts and Sciences of the City College of the City University of New York. While an undergraduate he served as chancellor of Sigma Alpha, the honor service society, and held a New York State Regents Scholarship. In June 1965 he received the degree Bachelor of Science cum laude. He was awarded the F. W. Lanchester Fellowship of Cornell Aeronautical Laboratory to pursue graduate study in the field of Operations Research, and entered the Graduate School of Cornell University in the fall of 1965. While a graduate student, Mr. Morgan served as a research and teaching assistant, as President of the Cornell Chapter of the Association for Computing Machinery, and was elected to Phi Kappa Phi. He is a member of the American Association for the Advancement of Science, the Association for Computing Machinery, The Operations Research Society of America, and The Institute of Management Sciences. He has published in the Technical Report series of the Operations Research and Computer Science departments of Cornell University, and in the Communications of the Association for Computing Machinery.

DEDICATION

To Eleanor

For the past, present, and future.

TABLE OF CONTENTS


| Chapter | Page |
|---|---|
| 1. INTRODUCTION | 1 |
| 1.1 Contemporary Data Processing Languages | 4 |
| 1.2 Instructional Computing | 8 |
| 1.3 Shared Resource Conflicts | 13 |
| 2. DATA STRUCTURES, PROCESSING, AND INPUT/OUTPUT FEATURES | 15 |
| 2.1 Data Structures | 16 |
| 2.1.1 Variables and Arrays | 19 |
| 2.1.2 Records | 21 |
| 2.1.3 Files | 23 |
| 2.1.4 Constants | 25 |
| 2.2 Processing | 26 |
| 2.2.1 Standard Programming Features | 27 |
| 2.2.2 Text Processing | 30 |
| 2.2.3 Iteration Control and Subroutines | 31 |
| 2.3 Input and Output | 34 |
| 2.3.1 File Directed Operations | 36 |
| 2.3.2 Device Directed Operations | 40 |
| 2.3.3 Formatting of Output | 43 |
| 3. FEATURES FOR ONLINE AND MANAGEMENT INFORMATION SYSTEMS | 46 |
| 3.1 Interrupt Blocks | 48 |
| 3.2 Remote Terminals | 52 |
| 3.3 Supervisor Features | 54 |
| 3.3.1 Interrupt Block Scheduling | 56 |
| 3.3.2 File Tagging | 63 |
| 3.4 An Organization for Management Information Systems | 65 |
| 4. IMPLEMENTATION | 69 |
| 4.1 Overall Program and Interface Structure | 70 |
| 4.2 Interrupt Monitoring | 77 |
| 4.3 File Handling | 81 |
| 5. SUMMARY AND DISCUSSION | 83 |
| 5.1 Instruction in Contemporary Data Processing Systems | 84 |
| 5.2 Generalizations and Extensions | 87 |
| 5.3 Discussion | 90 |
| 5.4 Summary and Conclusions | 92 |
| Appendix A: Summary of the DPL Language | 93 |
| Appendix B: Details of Implementation | 100 |
| References | 113 |

## LIST OF ILLUSTRATIONS


| Figure | | Page |
|---|---|---|
| 1. | The Statements of DPL. | 5 |
| 2. | Sample Data Description Section. | 18 |
| 3. | Information System Organization. | 66 |
| 4. | The DPL Compiler and Execution Monitor. | 71 |
| 5. | Sample Symbol Table Entry. | 73 |
| 6. | Sample File Description Block. | 73 |
| 7. | Example of translation to metacode. | 74 |
| 8. | Interrupt Block Scheduling Structures. | 79 |
| 9. | File Control Block (FCB). | 82 |

ACKNOWLEDGEMENTS

The author wishes to acknowledge the guidance and encouragement which has been given to him in the past three years by his friend and advisor, Professor William L. Maxwell of the Department of Operations Research. Professor Richard W. Conway of the Department of Computer Science has also provided invaluable assistance during this period. The author is grateful to Professor Lionel Weiss for his help in the field of Statistics.

Professors Conway, Maxwell, Sidney Saltzman, and Visiting Professor Howard Krasnow all contributed to the design of the DPL source language. Professors Conway and Maxwell, and Stephen Kennedy, a graduate student, played key roles in the implementation of the DPL compiler.

The design of the DPL system has been influenced greatly by the author's colleagues on the CUPL Project, Howard Elder and George Blomgren. The author is especially indebted to Howard Elder for writing the computer text editing system with which this thesis was printed.

The financial assistance given the author by the Cornell Aeronautical Laboratory, the Cornell University Department of Operations Research, the Department of Computer Science, and the National Science Foundation made the writing of DPL possible.

The author is grateful to the Cornell University Office of Computer Services for the help given to him both in the implementation of the DPL compiler and in the printing of this dissertation.

It is not possible to properly acknowledge the contributions made to all phases of this work by the author's wife, Eleanor Morgan.

# DPL: A LANGUAGE FOR INSTRUCTION IN CONTEMPORARY DATA PROCESSING CONCEPTS

## 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. These languages obscure basic concepts with operational detail and lack facilities to illustrate contemporary systems design. The Data Processing Language (DPL) is designed to:

1. teach the basic concepts of data processing necessary to understand the design, programming and operation of both conventional batch processing systems and of online inquiry and management information systems
2. serve as a test language for a new technique which may be used to organize and program systems which share a common data base among several users

DPL is not aimed at training coders or programmers, but rather hopes to enable students and future managers to understand how large computer systems are organized so that these people may carry on a meaningful dialogue with the computer professionals who will actually design and program the real world systems. The concepts which these people should know to fulfill the above objectives are discussed in Section 1.1.

DPL 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 new organization proposed for the handling of large shared data base systems may, however, be adapted for use with existing or future production languages.) The language processor includes extensive diagnostic aids as well as error correction features. This type of processor has proven worthwhile in teaching scientific computing but has not yet been tried with a data processing language.

Many other student oriented programming languages and compilers have been developed in recent years. A general discussion of the criteria for instructional computing systems is presented in Section 1.2. This section also contains a brief review of some of the instructional languages and the rationale for not using any of the existing data processing languages for instructional purposes.

A survey of the work on the resolution of conflicts among users competing for a shared resource is presented in Section 1.3. These conflicts arise in multiprogramming and multiprocessing systems, and may also occur with the new technique which is proposed in this dissertation for handling shared data base systems.

# DPL: A LANGUAGE FOR INSTRUCTION IN CONTEMPORARY DATA PROCESSING CONCEPTS

Howard Lee Morgan, Ph.D.

Cornell University, 1968

Abstract

The Data Processing Language (DPL) is designed with two aims. The first is to aid in teaching the concepts and techniques of contemporary data processing systems to those who need an appreciation of the field, but who do not need to become trained programmers. The second is to test a new method for organizing and programming large systems which share a common data base among several simultaneous users.

The criteria for a contemporary data processing language are set forth and DPL is shown to meet them. These include remote terminal management, handling of shared data bases, file and device oriented input and output, and standard arithmetic and text processing features. In addition, simple syntax and extensive error detection and correction features fulfill important requirements of instructional computing systems.

Interrupts are the basis for the new method of systems organization. The DPL programmer can specify conditions when interrupts should be generated, e.g., when the relation X + Y = 34 is true, and can specify the routine which should be called to process each interrupt. The DPL monitoring system detects the occurrence of these conditions and generates the interrupts. Some of the variables involved in conditions which can generate interrupts may be in files. The programmer can attach to those

files the interrupt processing routines and the interrupt conditions. When the file is read in, the system begins monitoring these attached interrupt conditions and may execute the attached interrupt processing routines, which are called file tags, even though the user who placed the tags on the file is no longer in control. Several interrupts may be generated as the result of the execution of a single LPL statement, and interrupts may be generated while executing an interrupt processing routine. Therefore, an algorithm is presented which schedules the execution of the interrupt processing routines.

A management information system may thus be composed of two parts: a data base of tagged files, and a supervisor program which handles interaction with remote terminals and performs background tasks. This new organization is shown to have value both for its instructional clarity and for designing and programming large integrated information systems.

Chapter 2 describes the data, program, and input/output structures of DPL. The features described there are sufficient to teach conventional batch processing systems design. The additional concepts needed for contemporary systems with online interaction are discussed in Chapter 3. This chapter also details the new organization for management information systems which is the heart of this dissertation.

In instructional computing systems, the language processor is almost as important as the language itself. Chapter 4 presents a brief picture of the implementation tactics followed in the DPL compiler and execution monitor.

Chapter 5 contains some extensions of the work presented in Chapter 3, a discussion of the relationship between data processing languages and simulation languages, and a summary and conclusions section which includes the author's views as to the usefulness and applicability of this work.

## 1.1 Contemporary Data Processing Languages

The concepts which are necessary in a contemporary data processing language include:

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

Any modern language for business data processing use must have the facilities to allow all of the above concepts to be expressed. An additional design criterion for DPL was that of expressing these concepts in as simple and straight-forward a manner as possible. Figure 1 provides a list of the DPL statements. The extra complexities and generalizations present in production data processing languages have been left out of DPL.

The first three concepts listed above are those which are necessary for the programming of conventional batch systems. These systems are well described in the introductory books by Gregory and Van Horn [26,27], Canning [5], Saxon and steyer [4], and McCarthy, McCarthy, and Hume [38]. Canning's book is especially recommended for its discussion of the manager's view of business data processing.

```
LET variable = expression
GO TO label
IF condition THEN statement ELSE statement
PERFORM 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. The Statements of DPL.

The fourth and fifth concepts listed relate to features needed in modern data processing systems, e.g., online inquiry and management information systems. Excellent discussions of these concepts may be found in Head [28], Desmonde [15], and Dearden and McFarlan [13]. Head's discussion of the American Airlines SABRE system provides much insight into the problems of a large system and focuses attention on the concepts which must be understood before such a system can be programmed. Detailed descriptions of some of the program organizations found in other online systems may be found in Martin [41].

None of the above mentioned books, with the exception of Head, devotes much space to the problem of selecting a language in which to program the system described. Most of the advanced systems mentioned were programmed in assembly level languages because of core constraints, object time constraints, or the simple lack of any suitable high level language.

The most widely available language for data processing is unquestionably COBOL [6]. A COBOL language processor is now required on all machines purchased by the U.S. government for business data processing purposes. The drawbacks to using COBOL as an instructional tool are discussed in Section 1.3.

Various computer manufacturers and computer users have recently been developing other languages for file manipulation. These languages include the IBM Generalized

Information System, GIS [23], GE's Integrated Data Store, IDS [31], and the General Motor's developed Associative Programming Language, APL [2] (which is in no way related to the Iverson language of the same acronym, APL [34]).

There have also been several recent developments in general purpose languages. IBM's introduction of PL/I [47], which is designed to serve the needs of scientific, commercial, list processing and string processing users alike caused great hopes to rise. The implementations have so far proven rather inefficient, which is delaying acceptance of the language. BCL [29], developed in England, is interesting for its new approach to definitional facilities. The data and program sections are intermixed, and the definitional facilities may even be used at object time.

None of the above mentioned languages was developed with instructional use in mind, and all have several drawbacks to use in this manner. These are discussed in the next section.

## 1.2 Instructional Computing

Some of the criteria which are used in the design of an instructional computing language or processor are:

1. easy to learn
2. unity concept/statement ratio
3. fast compile and go system
4. simplified I/O
5. extensive and easily understood diagnostics
6. error correction

That an instructional language should be easy to learn is agreed upon by almost everyone. The concept/statement type ratio (CST ratio) serves as a measure of the instructional efficiency of a language. It is defined as the number of concepts which can be easily expressed in a given language divided by the number of different statement types in that language. FORTRAN [20], for example, has two different statement types used for assignment (ASSIGN N TO K and K=N). Therefore, if this were the only concept expressed in the language, the CST ratio would be 1/2. In DPL, the CST ratio for assignment is 1. It is also possible to find cases where the ratio is above 1, if for example, there are a number of permissible modifiers on a statement type and these really illustrate several concepts. It seems fair to say, however, that the closer to unity the CST ratio of a language is, the more efficient that language is when it comes to instruction, since with each statement type, only a single concept is introduced. The actual calculation of CST ratios for real languages is an extremely subjective procedure, as the definition of the concepts which are illustrated by a particular statement type is a difficult task indeed on which to get agreement.

In the past decade, many languages and language processors have been developed which fulfill some or all of the above criteria. These language processors have been in the areas of scientific computing and simulation, and not in the area of data processing.

There are several language processors available for FORTRAN, each of which fulfill the third, fourth, and fifth criteria on the list, but do not attempt to meet the other criteria, preferring to accept the FORTRAN source language as sacred. These processors, namely, PUFFT [50], WATFOR [54], and DITRAN [42], have been widely used by universities to reduce the running time of student FORTRAN jobs. The problem of assisting in the instruction of FORTRAN have not, however, been attacked by these processors.

In 1962 the first language to fulfill all six of the criteria was developed. This was CORC [10], the Cornell Computing Language. It was the first language to perform error correction, and the payoff of this feature should not be underestimated. Studies of CORC[22], showed a significant decrease in the number of machine approaches 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 to run correctly. In late 1964, an instructional simulation language based on SIMSCRIPT [39] was developed using CORC as a base. This language, CLP [9], proved as successful in teaching simulation as CORC had in teaching scientific computing.

There were several other languages which fulfilled all of the criteria except for error correction. Some of these have enjoyed very wide acceptance. MAD [1], written at the University of Michigan, was one of the most used languages on the IBM 7090 machines. Three languages have also enjoyed success as time sharing languages, namely, BASIC [36], APL [34], and IITRAN [4].

In 1966 the CUPL [58] language was developed for the 360 computers. Into this design went the experience of four years work with CORC. This language has error correction features and is now in use at Cornell University.

DPL is the first language which not only satisfies all of the criteria mentioned in this section for an instructional language, but also satisfies all of the needs of a data processing language as discussed in Section 1.1. DPL includes parts of CUPL, most notably the algebraic manipulation and other standard programming features. Certain features which are not often used in business applications (such as trigonometric functions) remain in the compiler but are not stressed in instruction. Other features of CUPL, such as matrix algebra, have been removed for economy of space.

Is a new language needed for data processing instruction? Let us examine the alternative languages. COBOL, the most widely used data processing language, falls short in several areas. First, the verbosity of the language, as reflected in the low CST ratio, interferes with understanding which parts of a program are important and which parts are not. Second, the great flexibility in file, record, editing, field, and access method specification cause the beginner to be caught up in a morass of detail without having seen the overall structure. Recently, in fact, several private software companies have begun to market programs which allow the user to code in a special shorthand on several different special forms and produce COBOL programs as output. These packages (COGENT [7], MARK/IV [48]), would never have been needed if it were not for the burdensome nature of writing COBOL programs.

The third major problem with COBOL is that it does not have all of the facilities needed to write a modern system with online terminal access. This situation is being corrected, and there are efforts to add these fundamentally new features to the language [49], but the increased capabilities are gained only at the expense of an even greater increase in complexity.

While PL/I does not suffer from the CST ratio problems which COBOL has, the sheer number of different things which can be done with the language acts as a deterrent to instructional use. Also, the current IBM implementations of PL/I do not meet the third, fourth, or fifth criteria for this type of language.

Since the BCL approach is basically different from the standard one which is used in DPL, it does not appear to be a direct competitor.

## 1.3 Shared Resource Conflicts

Modern multiprogramming and multiprocessing systems such as MULTICS [3] or those described by Parkhill [46], usually contain provisions which allow several users to share a data base, i.e., access a common data base whether or not they are multiprogrammed with another user of that same data base. Ideally, the individual programmer would like to ignore the fact that the people with whom he is sharing the data base may be multiprogrammed with him. If he does, conflicts may arise which must be either prevented by the system from causing trouble, or accepted as a risk by the users.

The typical conflict situation arises when two users each read a piece of data from the shared data base and use that value in some calculations which will update the same or another member of the data base. For example, suppose that users A and B each wish to add 1 to the current value of X, which is in the shared data base, thus yielding the new value X+2. If user A reads the value of X, adds one, and, before user A stores the new result user B reads the value of X to add one to it, the final result of the two additions will be X+1, and not the desired X+2.

Various lockout and signalling mechanisms have been proposed to alleviate this problem. Work has been reported by Dijkstra [16], Van Horn [14], and in unpublished discussions on the Burrough's B500. Most of these techniques fall into one of two classes.

The first class, typified by Van Horn, has the system do all of the checking, and forces the system to lockout any data item from all other users of the data base whenever one user has read it with the intention of writing that item.

The second class uses a sort of caveat emptor rule and puts on the user the burden of locking out other users if he is doing sensitive calculations. The work done in simulation languages uses this class of rules. See, for example, GPSS-III [24].

The DPL approach is that of Van Horn, i.e., the system takes the responsibility. This is discussed more fully in Chapter 3 in the section on interrupt block scheduling.

## 2. DATA STRUCTURES, PROCESSING AND INPUT/OUTPUT FEATURES

This chapter details the handling of data structures, arithmetic and text processing, and input/output in DPL. A knowledge of the data and program structures is necessary for a proper understanding of the way that management information system concepts, presented in Chapter 3, are expressed in DPL. Since a complete summary of the syntax and semantics of DPL is given in Appendix A, the following discussion will not attempt to be all inclusive, but will present the important concepts of how data and program structures are named and used in a DPL program.

## 2.1 Data Structures

In the business data processing environment, large quantities of information are rendered manageable by grouping them into hierarchical structures and using common routines to process logically similar groups of information. DPL provides for the three level hierarchy commonly used in data processing languages [6,47]. These three levels are:

1. files
2. records
3. variables and arrays

Variables (or arrays, which are a data structure at the level of variables) contain the specific information, i.e., values, which are processed, e.g., names, addresses, amounts, etc. Records are composed of groups of variables and provide a way of naming a logical group of variables, e.g., an employee number, employee name, weekly salary, and year to date earnings may be combined into an employee record. This logical group may be moved, read, or written as a unit, even though processing may deal only with individual elements of the record. Files are in turn composed of groups of records, usually containing similar information, e.g., all of the employee records for a company will comprise the employee file. Files are treated as a unit by high level data management systems for control and other purposes, e.g., control of generation numbers, backup, etc. Not all of the variables in a program will be parts of records or files. Some variables may be used for working storage.

At the beginning of each DPL program, the programmer provides a Data Description Section which specifies the format of the hierarchy desired, as well as presenting specific information about the type and form of variables. A sample section is presented in Figure 2, and all references to data items in the rest of this chapter appear in this sample Data Description Section. In contrast to the data definition sections provided in COBOL, storage is not assigned for records or files by the description, but is only assigned for those variables or arrays which are not logically connected to any record. This is similar to the use of a dummy section (DSECT) in 360 Assembler Language [55], and has recently been proposed as an addition to COBOL in the form of a structure section [49]. In DPL, storage is assigned for the higher level structures only through the execution of CREATE or OPEN statements, which are described in Sections 2.1.2 and 2.1.3, respectively.

* MORGAN,HOWARD TEST PROGRAM

| FILE TYPE | ACCESS[1] METHOD | HDR[2] REC | RECORD TYPE | KEY[3] REC | VARIABLE NAME | SUBSCRIPTS 1ST | 2ND | FORM[4] |
|---|---|---|---|---|---|---|---|---|
| | | | | | I | | | N |
| | | | | | J | | | N |
| | | | | | TOTEARN | | | N |
| | | | | | TABLE | 30 | 5 | N |
| | | | | | TEXTVAR | | | T |
| | | | | | TXTARRAY | 12 | | T |
| | | | | | LIST | 30 | | R |
| | | | | | PRI | | | R |
| | | | | | OLDMASTR | | | R |
| | | | | | NEWMASTR | | | R |
| | | | | | DEV2 | | | R |
| | | | | | DEVB | | | R |
| PAYFILE | SQ | H | HDR | | KIND | | | T |
| | | | | | DATE | 2 | | T |
| | | | PAYREC | | CLOCKNUM | | | T |
| | | | | | NAME | 2 | | T |
| | | | | | CUMEARN | | | N |
| | | | OVERTIME | | AMOUNT | | | N |
| LABDISTN | DA | H | HDR | | | | | |
| | | | ACCOUNT | K | ACCTNUM | | | N |
| | | | | | AMT | | | N |

[1] Sequential (SQ) or Direct access (DA).
[2] 'H' for one record type per file. Indicates header record type.
[3] 'K' for one variable for each DA file. Specifies the key variable.
[4] Specifies the form of the variable. N=number, T=text, R=reference. Default assumption is number.

Figure 2. Sample Data Description Section.

### 2.1.1 Variables and Arrays

A simple variable is the smallest unit of information in DPL and may contain only a single value. The information may be a representation of one of three forms of data, which are:

1. number
2. text
3. reference

The <u>number</u> form has as its value a floating point number carried on the 360 as a double word floating point number. When the system requires integer values, automatic relative roundoff routines are invoked to resolve the roundoff problems usually associated with floating point computations [60]. This single number form eliminates the mixed mode problems found in many programming languages [20], and allows the beginner to concentrate on other, more important problems.

The <u>text</u> form has as its value eight characters of alphanumeric information, stored left justified and padded with blanks. This value may be printed as a character string, or used in computations.

The <u>reference</u> form contains a piece of information about the data structure as its value. This form is used, for example, for variables which will point to a particular instance of a record structure or a file structure. The internal form of this variable is a relative address which is intolligible 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 in a DPL program by its name, e.g., J, TEXTVAR.

An array, like a simple variable, contains actual information rather than providing structural convenience, but is different in that it may contain more than one value. Each value is called an element of the array, and single values are selected by writing the array name followed by the one or two dimensional coordinates of that element within the array, enclosed in parentheses. The fifth element of the array LIST, for example, is written as LIST(5), a notation common to almost all programming languages [20].

An array is declared in the Data Description Section, along with the form of the array (which is the same for every element) and the maximum values for each coordinate of the array, which must be positive integer values.

Only single elements of arrays may be used in computations in DPL (although this is not always true in programming languages, e.g., BASIC or CUPL matrix algebra [36,58]). Hence, when an array name is mentioned in a computation, it must be followed by the required number of subscripts, e.g., LIST(I), TABLE(2,I+3).

### 2.1.2 Records

A record is unlike a simple variable in the following important respect. When a variable is declared (and is not part of a record), its "name" is assigned and storage space is allocated to contain the value which that name represents. When a record type is declared in the Data Description Section, neither a name nor storage space are assigned for either the record or the variables which make up the record. Both of these assignments are accomplished during the execution of the program through the CREATE statement.

For example, the record type PAYREC is declared in the sample Data Description Section to consist of three variables or arrays, namely, the variable CLOCKNUM, the array NAME, and the variable CUMEARN. When the statement:

CREATE PAYREC REF PR1

is executed, four locations (double words) in storage are set aside to be a PAYREC, and the name PR1 is assigned to this particular PAYREC. When the programmer wishes to refer to this particular PAYREC for processing he does so by name, i.e. PR1, and not by record type, i.e. PAYREC. In general there will be several similar CREATE statements, each creating a PAYREC, but each assigning a different name to the one created.

When the programmer wishes to use one of the variables in the record for computation, he refers to it by a compound identifier consisting of the record name, a period, and the variable name, e.g., PR1.CLOCKNUM, PR1.NAME(2). This notation was chosen to be consistent with the list structure and pointer notation of PL/I [47].

When one is finished using a particular instance of a record type, the space occupied by it may be released by executing a DESTROY statement, e.g.:

DESTROY PR1

This will free the storage occupied and cause any further references to elements of PR1 to be treated as errors.

There are numerous advantages to using a record instead of many variables. First, the logical structuring permits us to treat the group of information as a unit for purposes of input and output. Second, when there exist several instances of a particular record type, the computational statements need be written only once to process all instances by using an array for the reference variables, e.g., one may write LIST(J).CUMEARN, and vary J in order to get at different instances of PAYRECs.

### 2.1.3 Files

Files are the highest level of the DPL data hierarchy, and are composed of many records. The logical structure of a file, i.e. the ways that records within the file may be accessed, is a concept which every student of data processing must understand. In DPL there are three access methods, i.e., methods used to insert, retrieve or delete records from files, which may be used. These are sequential, direct access, and indexed sequential, this latter method really being a special case of the direct access method.

Files meant for use with any of the access methods must begin with a header record, which is specified in the Data Description Section. This is similar to the label on magnetic tape or disc files which is required by most operating systems [44]. Direct access files may contain only one type of record in addition to the header, and these records are logically placed in the file in ascending order of the key variable's value. For implementation reasons, the key variable must be a simple variable which is part of the data record, and cannot be an array element.

Sequential files may contain instances of many different record types. These files should normally close with an end-of-file record, but this is the programmer's responsibility. A better understanding of the need for different access methods and their use in DPL is provided in

Sections 2.3 and 2.3.1.

In the same way that a particular instance of a record type is given a name and storage space through a CREATE statement, an instance of a file type is named and allocated through execution of an OPEN statement. For example, the statements:

OPEN PAYFILE REF OLDMASTR USING -dsname-

OPEN PAYFILE REF NEWMASTR

would create two files of type PAYFILE, one of them a new one which contains no information until the program writes some data on it (NEWMASTR), and the other one (OLDMASTR), which would contain the information stored by the system under the name -dsname-. This USING option allows instructors to create test data files for their students, which they then can use by statements of the type shown above. The instructor uses a DPL program to generate the file and then instructs the system to store the data by saying:

CLOSE NEWMASTR SAVE -dsname-

This is also the analogue of the DESTROY statement for records and will free any internal storage used by the file.

Files are referred to only by their name as assigned through an OPEN statement. The elements which are in files, namely records, are tied to the files only through the READ and WRITE statements discussed in Section 2.3.

### 2.1.4 Constants

Constants are like variables in that they can only contain one value and may be used in computation, but they differ in that they can never change value and in the way that they are named. Essentially, the name of a constant is its value, which may be of either number or text form. The number form is written as either an integer, e.g., 2, 345, 5000, as a floating point number, e.g., 3.1415926, 234.789, 11.5, or in exponential notation, e.g., 3.2E-5, 42.2E3, where E has the meaning "times 10 to the power."

Text constants are written as strings of characters enclosed in single quotes ('). They may be of any length and may contain any character except a quote mark, e.g., 'FAIL', 'THIS IS TEXT', 'PAYFILE'. In WRITE statements text constants may be of any length. When used in computations, however, they must be no longer than eight characters.

## 2.2 Processing

The processing statements in DPL are very similar to, and in some cases identical with, those of the Cornell University Programming Language, CUPL. The CUPL manual [58], provides a concise and complete description of that language so only those features necessary to understand the management information system extensions will be detailed in this chapter.

The methods of algebraic manipulation, flow of control, and logical testing are the same as those of CUPL and are similar to those found in most algebraic languages. Section 2.2.2 on text processing represents a new feature which is not present in CUPL but is certainly needed for data processing, since much of the information consists of text. The basis for the interrupt structured management information system discussed in Chapter 3 is the subroutine and iteration control mechanism of DPL and CUPL, namely the BLOCK structure, which is discussed in Section 2.2.3.

### 2.2.1 Standard Programming Features

DPL shares with many other procedure oriented languages the basic ideas of assignment, flow of control, and alteration of the flow of control through logical testing. These features of DPL are shown here in order to familiarize the reader with DPL's particular notation.

The normal flow of control is to execute one statement after another, in the order in which they appear in the source program. This normal flow may be altered in three ways. First, a STOP statement may be executed to transfer control to the system and terminate the program. Second, control may be transferred to some other point in the program by executing a GO TO statement, of the form:

```
GO TO -label-
```

This specifies the simple or subscripted label of the statement to which control should be transferred.

Labels are defined by their appearance in the label field of a statement. Simple labels are referred to in the same way as simple variables, i.e., by their name. Subscripted labels are not always a feature of procedure oriented languages, being absent from both FORTRAN and CUPL, but permitted in CLP and APL [9,20,34]. The subscript for a label may be of either numeric or text form, and is written in the same way that an array subscript is. The statements:

```
SUBSLABL(34)       STOP
SUBSLABL('LAST')   GO TO SUBSLABL(I)
```

would define two instances of the subscripted label, SUBSLABL. If the simple variable in the second statement had the value 34, control would be transferred to the first statement. There may be up to ten different subscripts for a given subscripted label.

The third method of altering the flow of control is with a PERFORM statement, which branches to a subroutine, but has control returned to the statement following the PERFORM statement when the subroutine is completed. PERFORM statements are discussed more thoroughly in Section 2.2.3.

Assignment statements are of the form:

LET -variable- = -expression-

where -variable- stands for the name of a simple variable, array element, or part of a record, and -expression- is an expression of the same form (number, text, or reference) as the form of -variable-. The form of a reference expression is simple, as it must be a single reference variable. The form of text expressions is discussed in Section 2.2.2. Arithmetic expressions are written as in CUPL or FORTRAN, and the DPL standard functions are listed in Appendix A.

One of the most important techniques in programming is that of making logical tests and altering the flow of control on the basis of such tests. In DPL, the logical test statement is of the form:

IF -condition- THEN -s1- ELSE -s2-

Both -s1- and -s2- are statements of the language (except IF statements) and either the THEN clause or the ELSE clause, but not both, may be left out. The -condition- phrase is composed of expressions as described in the assignment statements, and the relational operators =, NE, LT, LE, GT, and GE. If the expressions on the two sides of a relation are not of the same form, the relation is considered false. Several groups of expressions and relations may be combined into a compound condition by using the connectives AND or OR. (To ease the problems of teaching precedence rules for AND and OR to beginners, either ANDs or ORs, but not both, may be used in a single condition.) An example of an IF statement which illustrates several of the above points is:

```
IF I=J AND TEXTVAR NE 'NOGOOD' THEN STOP ELSE LET I=I+1
```

### 2.2.2 Text Processing

As mentioned in Section 2.1.4, text constants in DPL are written as strings of characters enclosed in quotes, e.g., 'HELLO'. These may be used in IF, WRITE, LET, and PERFORM statements, and as subscripts for labels, e.g.,

```
LET TEXTVAR = 'HELLO'
IF TEXTVAR NE 'HELLO' THEN GO TO SUBSLABL('LAST')
```

Text information is stored in variables in groups of eight characters. These may be read and printed like any other values. Text arrays, which are defined like other arrays, however, are treated differently on input and output. The array TXTARRAY, which is listed as 12 elements long on the sample Data Description, may be set to a single continuous string of 96 characters on input, and may be printed as a continuous string of up to 96 characters on output, by leaving off the particular subscript value which is normally required. If, for example, TXTARRAY(1) had the value 'ABCDEFGH', and the rest of the array was all blanks, the statement:

```
WRITE TXTARRAY(1),TXTARRAY
```

would print eight characters for the first item, and 96 characters for the second item on the list.

Extensive string manipulation facilities, such as those found in SNOBOL [18], were not deemed to be of sufficient importance to the beginner to be included in DPL.

### 2.2.3 Iteration Control and Subroutines

Most procedure oriented languages have some mechanism for controlling iteration, e.g., FORTRAN DO loops, PL/I DO clause, ALGOL for...step...until clauses, and a separate mechanism for creating and linking to subroutines, e.g., CALL in FORTRAN or PERFORM in PL/I. In the CORC language written in the early 1960's, iteration control and subroutine linkage were combined into a single concept which has proven easier to teach to beginners [10]. This method has been retained in both CUPL and DPL.

All code which is meant to be part of a loop or a subroutine is written as a block. A block consists of a sequence of statements delimited by the two statements:

```
-blabel- BLOCK
-blabel- END
```

where both instances of -blabel- are the same label (either simple or subscripted). This creates a subroutine structure which must be called from another part of the program. The block can be executed by executing the DPL statement:

```
PERFORM -blabel- [modifiers]
```

where -blabel- is the label of the block. The modifiers allow for both loop control and subroutine linkage.

For iteration in the FORTRAN DO loop sense, the modifiers which are used are illustrated by:

```
PERFORM -blabel- FOR -v-=e1- TO -e2- BY -e3-
PERFORM -blabel- -exp- TIMES
```

where all expressions are numeric. The first example provides a loop variables as in standard FORTRAN, and the second merely provides a loop count, which is often all that is required.

Two clauses which are similar to those in ALGOL are evidenced by the following examples:

PERFORM -blabel- WHILE -condition-

PERFORM -blabel FOR -v- = -e1-, -e2-, -en-

In the first case, which is similar to the ALGOL until phrase (merely by reversing the relation), the BLOCK will be repeatedly executed until the condition becomes false. In the second case, the block will be executed once for each value on the list of expressions.

All of the above mentioned PERFORM statements cause the named block to be executed when the PERFORM is executed. There is a special form of this statement, with the clause WHEN -condition-, which is not of the above type, and is used with the management information systems extensions discussed in Chapter 3.

DPL blocks differ in two important ways from the procedures of PL/I or the FORTRAN subroutines. First, the variables in a block are not local to that block, but are global to the program in which the block is imbedded. Second, there is no facility for transfer of parameters to the block in the calling statements. The use of dummy variables for local/global and parameter transfer purposes is mainly a coding technique, albeit an important one, and, as DPL is not aimed at creating coders, these features were not included in the language. The lack of this feature in both CORC and CUPL has not appeared to make it harder for students to progress to standard production languages such as FORTRAN.

## 2.3 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 the 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 recognizable form. A device directed operation is one which involves the transfer of information to or from a specified device. There need be no logical connection between the information transferred in successive operations on the same device. A conversion from machine recognizable to human useable form, or vice versa, usually occurs in a device directed operation. Conventional batch processing systems are mainly file directed, while online inquiry or management information systems are more evenly divided between file 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 printer (device directed). Then, an employee record would be read from the employee file (file directed) and selected information 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 printed. The logical structure of the file specifies the next record input to a read request, while the record to be next printed is specified by a characteristic of the device, i.e., 55 lines to the page are desired.

Every I/O statement in DPL specifies some bits of information (variables, arrays, records), 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 (line printer) -- the file or device must be explicitly referred to by the name given that file or device in the OPEN statement.

Characteristics unique to file directed operations are discussed in Section 2.3.1, while those characteristics which are used for device directed operations are discussed in Section 2.3.2. Section 2.3.3 contains a discussion of the possibilities of formatting printed output in DPL, which is a rather important subject in business data processing, as the clarity of the reports produced by the system will have a great impact on whether or not that system is used to the greatest benefit.

2.3.1 File Directed Operations

File directed operations involve reading or writing a specified record of a file. For sequential files, the record to be read or written next is specified by a position pointer associated with the file by the system. 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 concept which is important is the logical sequence. Naturally, a file must reside on some physical device (a disc in DPL), but the concept of a file implies a logical organization of a large amount of data, rather than the physical organization of the device on which that data resides. Some modern operating systems recognize this and partially or wholly remove from the programmer's consideration the precise physical location of his files [3,44].

For direct access files the value of the key variable is used to specify which record is to be read or written. Hence, a statement of the form:

READ -recref- WITH -keyexp- FROM -dafile-

will cause the record with a key value equal to that of -keyexp- (either numeric or text) to be read from the direct access file referred to as -dafile- and placed in the record named -recref-. It should also be noted that direct access files have their records arranged in ascending key order, so that 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 value. This allows a direct access file to be read sequentially as well as being read with the direct access technique.

For sequential files, the statement:

READ -recref- FROM -filref-

will cause the next record of the file named -filref- to be read into the locations comprising the record named -recref-.

In order to assist the user in reading from sequential files which may contain a mixture of record types, or from devices which may contain a mixture of record types along with unstructured data, the special function TYPE is provided. The argument of this function is a file or device reference, and the result is a character string whose value is the name of the record type which appears next on the specified device or file, so that the user knows what type of record to CREATE or use to contain the information which will next be input from the file or device. For example, after an OPEN on a file named OLDMASTR, TYPE(OLDMASTR) might have the value 'HDR' for the files in our sample Data Description Section. When combined with subscripted labels, this function can be a powerful aid to processing sequential files or devices having varying record types appearing on them. Many other languages circumvent this problem by allowing the user to reread the record under a different format (e.g., MAD's LOOK AT statement [1]) or allowing him to transmit the record to another location in core under format control (e.g., ENCODE and DECODE in FORTRAN 63[21]). Either of these alternative solutions is merely a more complicated way of being able to test the type of the next record before it is read, and hence DPL provides this feature directly. If there are no more records on the file or device, the string 'FAIL' is returned.

The programmer is informed of end-of-file and no-record-with-specified-key-found conditions through the TEST function, whose argument is a file reference and whose result is either the character string 'FAIL' or the string 'SUCCESS'. The 'FAIL' setting will be given for I/O errors other than the two mentioned above, but, except for these two conditions, standard error messages and correction procedures will be used by the system. This function can, of course, be tested in an IF statement, e.g.,

```
IF TEST(NEWMASTR)='FAIL' THEN GO TO ERREOF
```

To write on a file, the user specifies a record and the file to which the record is to be added. The position in which the record will be placed depends on whether or not the file is sequential or direct access. If the record is being written on a sequential file, it will be written at the place following the current pointer position, and the pointer advanced one record. On a direct access file, the record will be logically placed in the ascending key sequence. The normal form of a WRITE statement is:

```
WRITE -recref- IN -filref-
```

Often, the user will wish to ensure the safety of a particular file from being overwritten through his own carelessness. The statement:

PROTECT OLDMASTR

will make the specified file read only until the file is closed by a CLOSE statement, and any attempts to write on the file will not be honored.

The user may reposition the pointer associated with a file at any time by executing a POSITION statement. The forms of this statement are:

```
POSITION -filref- AT HEADER
                  AHEAD -exp RECORDS
                  BACK   -exp- RECORDS
```

which will reposition the pointer as indicated, whether the file is being used for input or output. If the command cannot be executed, (e.g., POSITION NEWMASTR 30 RECORDS AHEAD when pointing to the last record of the file) the closest command to the desired one will be executed, an error message will be given, and the TEST function will be set to 'FAIL'.

### 2.3.2 Device Directed Operations

Device directed I/O operations also specify lists of variables, records, or text information, and the device to or from which the transfer is directed. If devices other than the reader and the printer are desired, they must be described in 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 can accept input and output, and each of which can generate interrupts, whose handling is described in Chapter 3. 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 through an OPEN statement, e.g.:

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

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

The typical form of the read statement is:

```
READ -list- FROM -ref-
```

where -list- is either a single record reference or a list of simple variables, array elements, and text arrays, and -ref- is either a variable which points to a device or is left blank, indicating the card reader is to be used.

Some examples might be:

READ PR1 FROM DEVB

READ I,J,TEXTVAR FROM DEV2

READ TXTARRAY

In the first example, the next record on the device named DEVB is read into the record named PR1. On device directed input of entire records, the record type name must appear as a string at the beginning of the record. This may then be tested by means of the TYPE function described in Section 2.3.1. The second example would take successive values from the input stream of device named DEV2 and place them in I, J, and TEXTVAR, respectively. The third example would read from the card reader the text array TXTARRAY, in the manner indicated in Section 2.2.2. For all of the READ statements except the last, the success or failure of the operation can be tested with the TEST function, which is described in Section 2.3.1.

In device directed WRITE statements, a list of variables, arrays, and records is provided, along with some format control information, and the name of the device to which the operation is directed. If the output is to be stored magnetically, i.e., the destination of the data is not ultimately a printer, only values are stored. For example,

WRITE I, J, TEXTVAR ON DEV2

would write three values on device DEV2 which could later be read by the second READ example given above. If the device is one from which printed output is desired, the format controls play an important part. These are discussed in the next section.

### 2.3.3 Formatting of Output

When a device directed WRITE statement references a device which implies printing of output, e.g., line printer, teletype, the DPL system uses information provided in the WRITE statement to format the output. This format control information is provided in a manner simpler than the FORTRAN format statements or COBOL picture definitions, and yet is flexible enough to enable the student to learn the rudiments of report generation.

Output lines are made up of six fields, each of which may contain either a number, a text value, or the name of a variable, array, record, or file. When a WRITE is executed, successive elements are assigned to successive fields on the line, with some elements occupying only a single field, others more than one field.

The typical output statement is of the form:

WRITE -list- ON -devref-

where the device reference -devref- is optional and the list is composed of five types of elements, which are:

1. variable names, which print the name of the variable and the current value, in two successive fields, which cannot be split across two lines.
2. record names, which print the record type name followed by the values for all of the elements of the record.
3. messages, or strings of any length enclosed in

quotes as are text constants and print exactly as written, without the quote marks.

4. /variable name, which prints the value of the variable, but not the name, in one field.
5. $variable name, which prints the value of the variable, with a $ to the left of the most significant digit, and two decimal places to the right of the point, in one field (variable must be of number form).

Here is an example of how this scheme is used to print a report of earnings for all employees. The program:

```
WRITE '        REPORT OF EARNINGS'
WRITE 'EMPLOYEE NAME' , 'EARNINGS'
                ...calculate the earnings in a PAYREC
                called PR1
WRITE PR1.NAME, /PR1.CUMEARN
                loop for all PAYRECS and get totals
WRITE 'TOTAL', $TOTEARN
```

would produce the following output

```
          REPORT OF EARNINGS
EMPLOYEE NAME         EARNINGS
JONES, JOHN           245.00000
SMITH, SUSAN          300.00000

TOTAL                 $545.00
```

This shows how the simple formatting scheme can be used to print reasonably attractive reports. A facility similar to this is used in CUPL with much success (although CUPL does not have the $ option).

# 3. FEATURES FOR ONLINE AND MANAGEMENT INFORMATION SYSTEMS

A management information system is composed of four parts:

1. Data base -- all of the information which is available to the rest of the system.
2. Data entry and updating -- programs which are used to keep the data base current.
3. Inquiry -- programs which utilize the data base in a read-only manner, e.g., online inquiry systems.
4. Supervisor -- the program which schedules the execution of all other programs in the system on a "when needed" basis.

DPL is a language in which all four of these parts have representations simple enough to be easily grasped by the beginner. The data structures described in Chapter 2 allow the data base to be represented as a group of files. The data entry and updating programs may be written using the program structures discussed in the previous chapter. A new program structure, the interrupt block, is introduced in Section 3.1 in order to allow a means of handling remote input/output terminals such as those used in inquiry systems. This interrupt block structure also proves useful when writing supervisor programs.

Additional specialized concepts needed to program a supervisor are brought out in Sections 3.3.1 and 3.3.2 as extensions to the interrupt block structure. These concepts are used to develop a new organization for management information systems or other large file oriented systems. This new organization is especially suited to instructional use because of the clarity with which interconnections between the multitude of program functions and the data base are described. Section 3.4 details this organization and discusses its application to management information systems.

The value of this new approach in a real production context, and some extensions that might add to its value in both the instructional and the production contexts, are discussed in Chapter 5.

### 3.1 Interrupt Blocks

The growth in the use of interrupt mechanisms in computer hardware technology has paralleled the growth in sophistication of the software technology associated with monitor or operating systems. Interrupts were first used to provide a convenient means for the hardware to inform the monitor that certain events had occurred, e.g., completion of an I/O event or the value of a timer reaching zero [32]. The second stage in the use interrupts saw the class of monitored events broadened to include what were essentially software errors, e.g., division by zero or attempts to execute illegal operation codes [2]. When the IBM 360 series of computers was introduced, along with IBM's plans for still more comprehensive operating systems, a third stage in the use of interrupts was intitated [43,56]. This third stage added a new class of events which could cause interrupts to be generated, namely the execution of a special instruction (the "supervisor call"). The 360 computers also allowed many more of the software error interrupts (or "program check" interrupts) than did earlier machines.

Interrupts are handled in basically the same manner on all machines, regardless of the class of event causing the interrupt. When the event which causes the interrupt occurs, the current value of the instruction counter is saved in a fixed memory location and an unconditional transfer is made to another fixed location which contains a link to the routine which will process the interrupt. Nesting of interrupts, i.e., the occurrence of a second interrupt of a specified class while a first interrupt of that same class is being processed by the interrupt processing routine, can create quite complex problems. Hence, most systems have a provision for masking off, or blocking, an interrupt from performing the transfer of control indicated above if it is an interrupt of the same class. In these cases, a special instruction must be executed before the second interrupt can be taken, e.g., ION ("interrupt on") for the PDP-8 [53].

Until the third stage in the use of interrupts began, essentially all interrupt processing routines were part of the monitor system, and were inaccessible to the programmer who was not writing in assembly level language. With the increased sophistication provided in third generation hardware and software, the PL/I language designers were able to allow the programmer to define his own interrupt processing routines in the high level language [47].

The PL/I user is able to specify the processing routines for most of the software error conditions which the 360 hardware monitors. He does this by writing statements of the form:

ON FLOATINGDIVIDE -procedurename-

This procedure is then performed whenever a floating point divide exception occurs. These procedures are called interrupt function modules [59]. The PL/I designers have provided for simulating some interrupts by means of a SIGNAL statement, i.e. when a SIGNAL statement is executed, the condition which appears in that statement is considered to have occurred and an interrupt is generated.

DPL introduces a fourth stage in the growth of the use of interrupt mechanisms by allowing the programmer to specify rather complex events whose occurrence will cause interrupts to be generated. The two classes of interrupt causing events which are felt to be most useful for management information system purposes are device interrupts, e.g., the "attention" key on a teletype, and program controlled interrupts described below. The interrupt function modules used to process the interrupts are written as DPL blocks, as described in Section 2.2.3.

The programmer indicates which block is to be used to process a device interrupt by executing a statement of the form:

```
PERFORM -blabel- WHEN -devref- INTERRUPTS
```

where -blabel- is the label of the interrupt processing block and -devref- is the name of the device which will generate the interrupt.

The program controlled interrupts are a powerful, new feature introduced in DPL. The programmer can cause an interrupt to be generated and an appropriate routine to be executed whenever a Boolean condition becomes true. These conditions can be complex combinations of variables and constants. For example:

PERFORM -blabel- WHEN PR1.CUMEARN GE X

would instruct the system to begin monitoring for the condition where PR1.CUMEARN is greater than or equal to X, and, when that conditon becomes true, to execute the block named -blabel-.

DPL performs the monitoring for these events or conditions through software checking, but this monitoring could conceivably be done by some form of microprogram or by hardware directly [45].

The monitoring may be terminated at any time by issuing the statement:

CANCEL -blabel- WHEN -condition-

where the condition has previously been issued in a PERFORM...WHEN statement.

This program controlled interrupt feature was implied in some early work on a language called BCL, written by David Hendry [29], but was never actually carried through in subsequent design and implementation. Hendry included a feature similar to the PL/I ON SUBSCRIPTRANGE statement, yet left the syntax free to contain instances similar to those allowed in DPL. Hendry's work was done at about the same time as the present work.

Later sections in this chapter will detail the handling of the complex conditions which can occur when many of these PERFORM...WHEN statements have been issued and complex interrupt nesting occurs.

### 3.2 Remote Terminals

The use of remote terminals in modern computing systems is becoming more and more frequent. Such applications as online inquiry systems, inventory control systems, management information systems, and management gaming can all make effective use of remote terminals. SABRE, the American Airlines reservations systems, is one example of a system in which remote terminal access to a data base is the *raison d'etre* for the system [28].

Devices are defined by a device declaration statement as shown in Section 2.3.2, and devices may be capable of performing up to three functions; input, output, and generating interrupts. The device directed I/O described in Sections 2.3.2 and 2.3.3 is normally used when communicating with remote terminals.

The concepts which a student of data processing should be familiar with in order to understand the use of remote terminals in modern systems include:

- polling
- interrupt processing
- lockouts
- man-machine interaction

The first two concepts are important in systems which expect a large number of remote terminals to be accessing and possibly updating a data base at the same time. The program must have some means for determining which terminal requires service at a given time.

A DPL program can poll terminals by issuing READ requests to the terminal and then using the TEST function to determine whether or not the request was satisfied. If it was not, the terminal does not require service. If data was input in response to the request, the terminal does require service by the program.

The second solution to the problem of determining which terminals require service is through the use of interrupts. Here the terminal user plays an active role in requesting service by pressing a key (labeled variously "attention," "escape," or "break") which will generate an interrupt to the program. As discussed in the previous section, the programmer can specify a processing routine for a particular device interrupt with the PERFORM...WHEN statement.

When there are a large number of terminals accessing a data base, the problem of lockouts arises. The situation which might arise would have two terminals, each updating the same variable, with the wrong result being saved. There has been considerable thought given to this problem, most notably by Dijkstra [16] and Van Horn [14].

The last concept which must be understood is that of man-machine interaction. Questions of this sort have a major influence on the dialogue that the system will carry on with the terminal user. The book by Martin [41] on real time systems design provide a valuable guide to this area.

### 3.3 Supervisor Features

A typical integrated management information system may include several groups of programs, each of which may require online access to the data base and may interact with any of the other groups. For example, there may be separate groups of programs to process payroll, inventory control, order entry, and general ledger accounting, all as part of a single integrated system.

The supervisor must know what conditions require the execution of each of these other programs, and the supervisor must also have some means of detecting the occurrence of these conditions so that it can schedule the running of these other programs. In DPL, interrupts are used to indicate the occurrence of these conditions, the interrupt block structure and PERFORM...WHEN statements are used to specify which programs process which interrupts, and the interrupt processing routines themselves are written as DPL blocks. The rules used by the DPL system to schedule PERFORM...WHEN blocks are given in Section 3.3.1.

The variables which are part of the conditions in the PERFORM...WHEN statements may be simple variables or parts of records. If the variables are in records, and the records are written onto files, evaluation of the conditions may require access to the files. DPL handles this situation through a new method called "file tagging," which is discussed in detail in Section 3.3.2. The implications of this situation play a major role in the new organization being proposed for management information systems and developed in Section 3.4.

### 3.3.1 Interrupt Block Scheduling

The execution of a PERFORM...WHEN statement causes the system to watch for the condition mentioned in the WHEN clause and, when that condition becomes true, to execute the interrupt block named in the statement. This is the program controlled interrupt feature of DPL.

When a PERFORM...WHEN is executed, the pair composed of the interrupt block name and the condition (denoted as the (b, c) pair) is placed on the "pending block" list (PB list). At the same time, a flag is set in the main symbol table (see Section 4.1 for a description of the symbol table) for all variables which are used in the condition. For example, if the statement:

```
PERFORM BLKC WHEN I-J+3=K
```

were executed, the pair (BLKC, I-J+3=K) would be placed on the PB list and the variables I, J, and K each would have a flag set in the symbol table.

The execution of a CANCEL...WHEN statement would remove the pair designated in the CANCEL statement from the PB list. If the indicated pair is not on the PB list, an error message is generated.

Whenever a statement which can assign a value to a variable is executed, e.g., LET, READ, PERFORM...FOR, the symbol table entry for that variable is examined. If that variable is flagged as being involved in some condition which is on the PB list, that condition is evaluated, along with any other conditions on the PB list in which that variable is involved. If any of the conditions have the value "true," the corresponding (b, c) pair is placed on the "to be executed" list (TBE list), and removed from the PB list.

The actual checking of conditions and generation of interrupts takes place upon the completion of the statement which performed the assignment. If a statement performs multiple assignments, e.g., READ X, Y, the evaluation of the affected conditions reflects all of the assignments. This is comparable to the doctrine on most machines that interrupts may only be accepted between instructions. At the level at which the DPL programmer writes, a DPL statement is equivalent to a machine instruction.

Note that the condition is checked only on store operations and is not checked at the time the PERFORM.. WHEN is executed and the (b, c) pair is added to the PB list. This is similar to the convention followed in some computer systems, namely: The execution of an interrupt enable command does not enable the interrupts until after the execution of the instruction following the interrupt enable instruction [53]. The reason for doing this in DPL is simple. It allows the last statement in an interrupt block to be a PERFORM...WHEN which will put the (b, c) pair for that block back on the PB list. Presumably, if the condition were checked upon execution of the PERFORM...WHEN statement, the condition would be true and a nesting problem would arise.

If there is only one (b, c) pair on the TBE list after an assignment statement has been executed, and the interrupt block named in it does not issue any PERFORM...WHEN statements, the block is performed and control is returned to the statement following the assignment statement. When there is more than one pair on the list, however, or some of the interrupt blocks issue PERFORM...WHEN statements, thus adding pairs to the PB list while an interrupt block is executing, the situation becomes quite complicated. The following examples may help to illustrate some of the problems which can arise, and will be used to show the rationale for the scheduling algorithm which was chosen.

1. Suppose the PB list contains the two entries (B1, X LE 5) and (B2, X LE 10). The statement: LET X=4 is executed. Which block should be executed first?
2. Suppose the PB list contains the two entries (B3, X=2) and (B4, X=2). Suppose further that execution of block B3 will not change the value of X, but execution of B4 will set X to 3 before exiting. Again, which block should be executed first?
3. In this example the PB list consists of the single entry (B5, X=3), and the first two statements in B5 are:

```
PERFORM B6 WHEN X=2
LET X=2
```

Should the execution of B5 continue after these two statements are executed, or should B6 be entered after execution of the LET statement?

Some of the problems arise from interaction of one interrupt block with the conditions on the PB list, and others arise from interactions between the PERFORM...WHEN statements, which may not all be consistent with each other. The execution of one block when more than one has been placed on the TBE list may change the condition which caused other blocks to be placed on the TBE list. Furthermore, the ordering of the execution of the blocks on the TBE list may affect the number of blocks which can be executed as a result of a single assignment statement.

The criteria used in developing the algorithm which schedules the execution of interrupt blocks were:

1. When an interrupt block is entered for execution, the associated condition must be true. (This may have been assumed when the block was written).
2. A unique ordering for the execution of the blocks must be guaranteed.
3. When several pending blocks are scheduled as the result of a single assignment statement, the execution of any one of these blocks should be transparent to all of the other blocks, i.e., each block may be written as if it will be the only block executed when an assignment is performed.

4. Nested interrupts should be treated with lower priority than those interrupts which are generated as a result of the initial store operation.
5. As far as possible, as many of the blocks which are initially placed on the TBE list should be executed.
6. The algorithm should be aware of any conflict situations which it cannot handle, and should report these to the programmer.

Algorithm A, presented below, meets all of these criteria and is the one used in DPL to schedule interrupt blocks.

ALGORITHM A

Step 1. Label all (b, c) pairs on the TBE list as level 1.

Step 2. Select from the TBE list all interrupt blocks whose execution will not change any of the conditions associated with blocks on the TBE list. That is, select those blocks which make read only accesses to variables which are involved in conditions on the TBE list. Execute the selected blocks in any order and remove from the TBE list .

Note 1: If any new interrupts are generated during the execution of any of the blocks executed in Step 2, add the new (b, c) pairs to the TBE list and mark these new blocks as level 2. Also remove these blocks from the PB list.

Note 2: If any PERFORM...WHEN statements are issued by any of the blocks executed, add the (b, c) pairs to the "reschedule" list, and not to the PB list.

Step 3. Test the condition, c, for the next (b, c) pair marked as level 1 on the TBE list. If c is true, go to Step 4. If c is false, go to Step 3. If the end of the TBE list has been reached, go to Step 5.

Step 4. Execute the selected block and remove the associated (b, c) pair from the TBE list. Notes 1 and 2 above apply to this execution. When execution is completed, go to Step 3.

Step 5. Flag all blocks remaining on the TBE list at level 1 as "in conflict," and remove these pairs from the TBE list, placing the pairs on the "reschedule" list. If the TBE list is now empty, go to Step 6. If not, go to Step 1.

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

It is instructive to examine the performance of this algorithm on the examples given above. In example 1, the pair which had been placed on the list first would be executed first. In the second example, the algorithm would first execute block B3, which desires only to read the value of X, and then would execute B4. B3 might, for example, be printing an exception report while B4 might actually take action on the exceptional condition. In example 3, the new (b, c) pair would not be placed on the PB list until B5 was exited, thereby eliminating the nesting problem.

Step 5 in Algorithm A, which flags conflicts in accordance with criterion 6, is a rather important feature of DPL's handling of interrupt blocks. If there is more than one programmer at work designing and programming parts of a large system, ambiguities in the description of the responses of the system will often arise. The algorithm actually tries to cope with these ambiguities, and only after all means of coping with the ambiguity have failed will it give up. Another example of this type of algorithm has recently appeared in the context of decision tables [35]. This bears a close relation to DPL's problems since the entire program controlled interrupt structure may be thought of as an asynchronous decision table processor.

In order to assist the programmer in using the program controlled interrupt feature, the WRITE ALL statement described in Appendix A can be used to output the current PB list.

### 3.3.2 File Tagging

Section 2.1.1 described the manner in which files are saved and made available to other users of the system, and the previous section has discussed the program controlled interrupt feature in detail. In this section, the two concepts are combined into a new feature called file tagging. File tagging can be used to simplify the effort which is normally associated with organizing and linking together the many programs which make up an integrated management information system.

Some of the variables which are mentioned in conditions which are on the PB list may be elements of records. For example, the pair (B2, PR1.CUMEARN GE 100000) may be on this list. The DPL system keeps a record of the file on which this record was last read from or written on, i.e., which file has the most current version of the value of the elements of this record. If the record is destroyed while the condition is still on the PB list, or if the file which contains the most current value of the record is closed with the SAVE option while the condition is still on the PB list, file tagging will take place.

While a file is tagged, the conditions which are associated with the elements of that file and the interrupt blocks which process those conditions are attached to the file , i.e., stored with the file. When that file is opened with the USING option, any conditions which are attached to the file are added to the PB list, just as if the user who opened the file had issued a series of PERFORM...WHEN statements. The initial user has, so to speak, put some tags on the file to indicate his interactions with elements of that file, whence the name, file tagging.

Several successive programmers may all put their tags on the file, the effect being cumulative. When the file is read in, all of the associated conditions are added to the current user's PB list.

A tag may be removed from a file by opening the file, issuing a CANCEL...WHEN statement for the condition which is no longer desired, and closing the file with the SAVE option.

Because of the global nature of variables in DPL, the use of strict coding conventions is mandatory when several programmers wish to use file tagging as a means of communication between programs.

One previous attempt to connect programs and files in the intimate manner described above was made in [37]. This scheme, however, was based purely on static decision table logic and was more suited to use as a means of production control than for use as a building block for large file oriented information systems.

## 3.4 A New Organization for Management Information Systems

The use of the file tagging and program controlled interrupt features allows a new organization for management systems. Rather than the four parts mentioned at the start of this chapter, the management information system would consist of two parts; a tagged data base and a supervisor program. The contrast between this form of organization and current methods of organizing systems is shown graphically in Figure 3. The new organization is closely related to parallel processing while the old reflects years of sequential processing experience.

In the parallel system, each program is written independently of all others, and is then attached to a file. The programmer also provides the supervisor with the PERFORM...WHEN statements needed to start the execution of his program. For example, the inventory file might be tagged with one set of programs by which the order entry requests are handled and a set of tags which would ensure that inventory control runs were made when the stock level dropped below a preset order point. In the sequential system, the order entry programmers would have to work intimately with the inventory control programmers to ensure everyone's satisfaction.

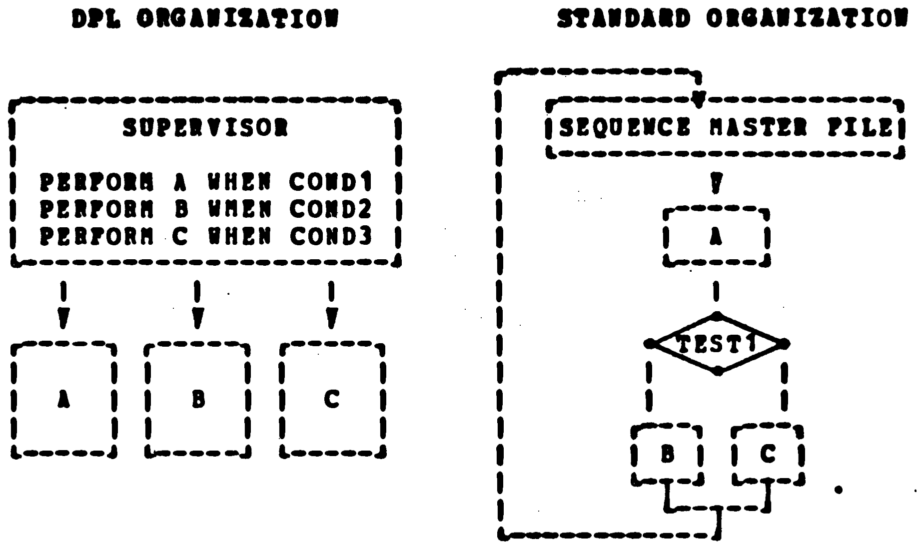



Figure 3. Information Systems Organization

The supervisor program would first open all of the files in the data base, thereby setting up a large PB list with all of the conditions which must be monitored. It would then open the remote terminals and either begin to poll them or accept device interrupts. Finally, the supervisor would begin the execution of some background task.

If one of the terminals interrupted and requested the program to update a particular file, that program would be executed, possibly setting off a chain of execution of other interrupt blocks. After processing was finished, the system would return to its background task.

The conflict recording mechanism of Algorithm A would indicate to a programmer when he was in conflict with some other programmer's requests for a file. These conflicts could then be handled manually between the programmers. What this amounts to is using the files as the main interface between programmers.

For instructional purposes, a class could be divided into several groups. The first group would have to design the files, and make their designs available to the rest of the class. Other groups would write the processing routines and the supervisor. All groups would specify their interactions with other groups by writing PERFORM...WHEN statements. Such an exercise would show the class how to separate the normal programming jobs such as file updating, report generation, etc., from the really complicated problems which arise in the interaction of tasks which require access to the same data base at the same time.

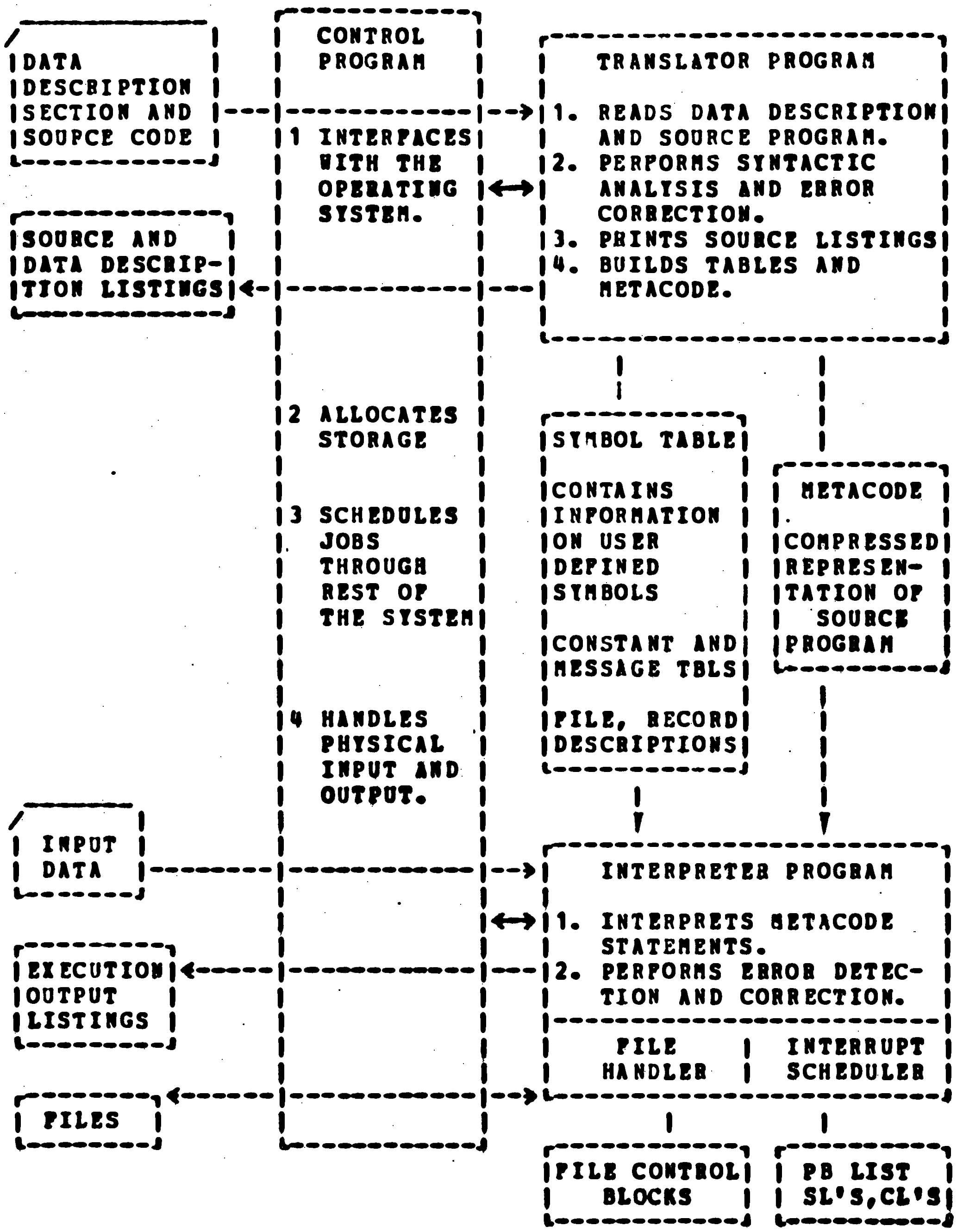
# 4. IMPLEMENTATION

The design of the language for an instructional computing language such as DPL is almost as important as the design of the language itself. This chapter describes briefly the implementation strategy used in constructing the DPL compiler and execution monitor system. Special emphasis is placed on the handling of the interrupt block features which are unique to the language. Details of the implementation, including descriptions of the programs, tables, and intermediate language are presented in Appendix B. The reader is referred to Feldman and Gries' excellent article on translator writing systems [19] for background material in compiler writing, and to Conway and Maxwell [11], and Freeman [22], for discussions of the error correction features as implemented in the CUPL and CORC language processors.

### 4.1 Overall Program and Interface Structure

The overall structure of the DPL processing system is shown in Figure 4. There are three major program segments; the translator, the interpreter, and the control program. Each of these programs is coded in a reentrant and self-relocating manner in 360 Basic Assembler Language. The interface between the translator and interpreter programs is composed of the symbol table and associated file and record description blocks, the constant and message tables, and the intermediate language representation of the source program, or "metacode."

The control program acts as the master scheduler for the system, provides common services to both the translator and interpreter, and handles all interfacing with the operating system under which DPL is running. This modularity with respect to operating system interfaces proves worthwhile when modifying the DPL system to run under several different operating systems. The scheduling aspects of the control program include the sequencing of each job through the translator and interpreter programs, the batching of many individual DPL programs into a single job with respect to the operating system (in order to reduce the overhead of loading the DPL system into core), and the monitoring of time-sharing users (although this last feature is not included in the current implementation). The services provided to the other programs in the system




——— FLOW OF CONTROL
----- FLOW OF DATA

Figure 4. The DPL Compiler and Execution Monitor.

include reading cards, printing output lines, error message formatting and printing, storage allocation, and file storage and retrieval. A complete list of control program services is presented in Appendix B.

The translator program reads the Data Description Section and the source program, performs syntactic analysis, and constructs the tables and metacode mentioned above. The translator corrects any syntactic errors which it finds, reports its corrections to the user, and makes the correction in the metacode, thus guaranteeing syntactically perfect metacode as input to the interpreter. A sample error correction report to the user might look as follows:

```
ERROR IN 0020 LET X=2I
DPL USES 0020 LET X = 2*I                    4A
```

The number at the end of the line is a reference to the DPL manual.

The symbol table contains information about all of the user defined symbols in a program, e.g., variables, labels, record types, record elements, etc. A sample symbol table entry for a simple variable is shown in Figure 5. When a record or file type is declared in the Data Description Section, a record or file description block (FDB or RDB) is created. This block, a sample of which is shown in Figure 6, acts as a description of the format of that file or record, i.e., which variables or records are permitted in the file or record, and in what order they appear. This is similar to the DSECT capability of 360 Assembly Language,

```
+----------------------------------+
|      NAME (1-8 CHARACTERS)       |
|----------------------------------|
|SIMPLE  |NUMERIC| ADDRESS OF CL   |
|VARIABLE|LHS/RHS|  FOR THIS VARIABLE|
|----------------------------------|
|       VALUE  (8 BYTES)           |
+----------------------------------+
```

Figure 5. Sample Symbol Table Entry.

```
+------------------------------+
|NUMBER OF|POINTER TO |POINTER TO|
|DIFFERENT|KEY ELEMENT|HEADER    |
|RECORD   |FOR DIRECT |RECORD    |
|TYPES    |ACCESS FILE|TYPE      |
|------------------------------|
|POINTERS TO EACH OF THE RECORD |
|TYPES ALLOWED IN THIS FILE     |
| (ONE BYTE PER POINTER)        |
+------------------------------+
```

Figure 6. Sample File Description Block.

which describes a block of storage, but does not allocate the storage.

The constant table and the message table are used to store all user defined number and text constants. These tables also contain information which allows the actual input format of the constant to be reconstructed, a feature useful for error correction purposes.

The metacode is a compressed representation of the source language program. Variables, labels, and constants are replaced by pointers to the symbol or constant table entries for these variables and constants, reserved words are replaced by tokens for these words, and some minor reordering of the elements of the source language occurs. A statement in both its source language and its metacode form appears in Figure 7.

```
Source code:    LET X = I + J

Metacode: F0 0020 C0 0121 80 0124 82 0126 F4
          Start line LET  X    =    I    +    J   END
                number
```

Note: Metacode is shown as a string of hexadecimal digits, stored in the 360 two per eight bit byte.

Figure 7. Example of translation to metacode.

This strategy of translating the source language into an intermediate language, rather than compiling actual machine code, has been used with much success in the CORC, CLP, and CUPL compilers. The metacode serves several purposes. First, metacode eases the task of correcting errors in the source program since all variables and labels are referred to through the symbol table. If the error correction routines decide, for example, that label ABC is really a misspelling of label ACB, the single correction to the symbol table entry will fix all references to ABC in metacode. If machine code were generated, and absolute addresses assigned, correction of this type of error becomes much more difficult. Second, the metacode provides a link back to the source language which the programmer entered. This is especially useful for error messages, since an error message which references source language variable names, etc., is far more meaningful than a message which refers to absolute addresses, as do certain FORTRAN messages. A special subprogram called the "reverse translator" produces source language statements from metacode, and is used to print messages such as the DPL USES example given earlier in this Section.

In CORC, CLP, and CUPL, a code generate phase followed translation, and generated machine code from the syntactically perfect metacode. (Of course, some semantic errors remained which were resolved only during execution.) In the DPL system, the metacode serves as the pseudo machine language of a "DPL machine," and in this form is executed by the interpreter program.

The interpreter uses the symbol and other tables as needed while interpreting the metacode program. Statement keywords are used as the operation codes, and interpreter subprograms are used to evaluate expressions, functions, and addresses. The interpreter also monitors for the occurrence of conditions on the PB list, and performs error correction for those semantic errors which occur, e.g., division by zero, or transfer into the range of a BLOCK.

The interrupt function handling routines and the file handling routines, both of which are subprograms of the interpreter, are discussed in the next two sections.

4.2 Interrupt Monitoring

The basic strategy in the implementation of the interrupt block features is the use of condition lists, referred to as CL's. Every variable which is mentioned in a condition on the PB list, be it simple, subscripted, or part of a record, owns an active (i.e., non-empty) CL. The CL is a list of pointers to all the conditions on the PB list which mention the owning variable. Therefore, whenever a variable is assigned a value, only the conditions pointed to by that variable's CL need be evaluated.

Simple variables, which have their own symbol table entry, have the pointer to their CL stored in their symbol table entry. Since each element of a subscripted variable, or each variable which is part of a record does not have its own unique symbol table entry (i.e., the entry covers a class of variables), an additional level of indirection is required to get to the CL. Each reference variable which points to a record, and each subscripted variable's symbol table entry points to a subscript, or SL. The SL is a list of those subscripts which have an associated CL, along with the pointer to that CL. For example, if AR(3) and AR(6) were involved in conditions, the SL for AR would contain the values 3 and 6, along with pointers to the CL's owned by AR(3) and AR(6).

Entries in the CL point to line numbers on the PB list. Each line of the PB list is an entry eight bytes long. The first two bytes point to the block which was named in the (b, c) pair by the PERFORM...WHEN statement which placed the pair on the list. The next two bytes are used by the interrupt block scheduler and tell whether or not the entry is on the TBE list, the "reschedule" list, or only on the PB list. The last four bytes point to the location in the intermediate language which is the start of the condition to be evaluated. If all of the variables in the condition were simple variables, the pointer to the intermediate language actually points to the metacode for the PERFORM...WHEN statement. If subscripted variables or record elements were used, a new copy of the condition, with all subscript expressions replaced by references to the constant which is equal to the value of that subscript expression at the time the PERFORM...WHEN statement is executed is used. Figure 8 on the next page shows some examples of the structures which result from the SL pointing to the CL which points to the PB list which points to the intermediate language.

The interrupt monitor is called immediately after each store operation. This routine determines whether or not the variable which is being assigned a value owns a CL. If it does own a CL, this CL is placed on a stack, a flag is set, and control is returned to the program which called the interrupt monitor. This may occur several times before the statement is completed, e.g., the statement READ X,Y,Z may cause three CLS to be placed on the stack.

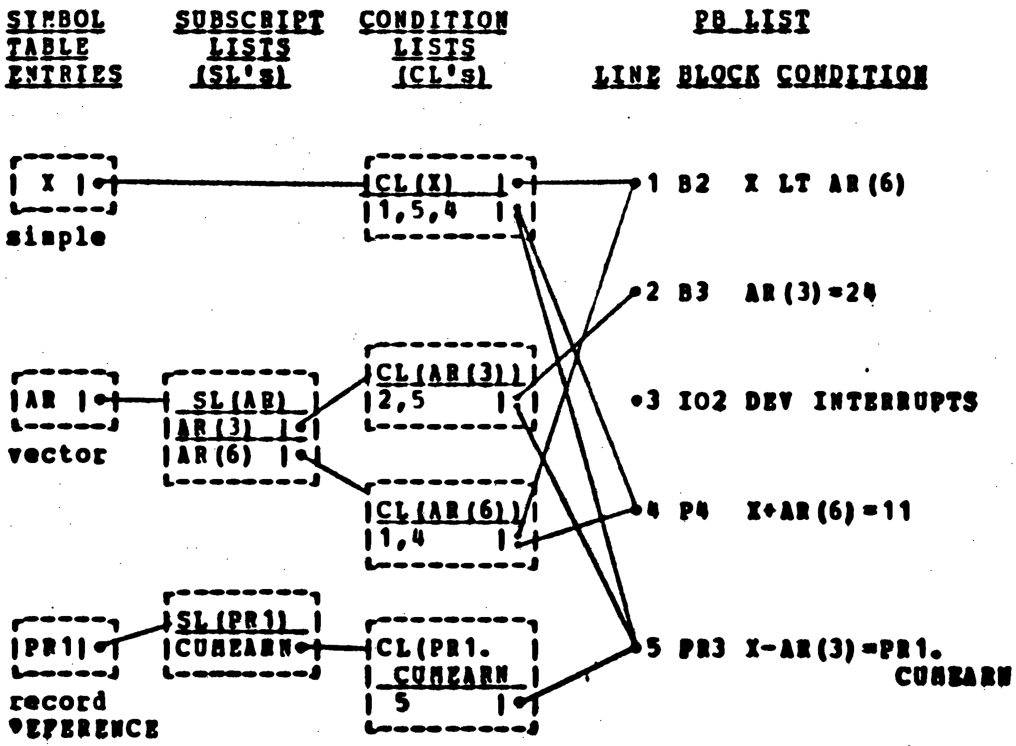



Figure 8. Interrupt Block Scheduling Structures.

The actual interrupt can only occur at the end of the entire DPL statement, since this is the level at which the DPL programmer writes, and is equivalent in his mind to the single instruction which an assembly language programmer writes. When control is returned to the interpreter for processing of the next statement, the flag which the interrupt monitor may set is interrogated. If the flag is on, indicating a store operation which has assigned a value to some variable involved in a condition on the PB list, the interrupt scheduler is called. This routine sets up the TBE list, checking all of the conditions indicated in the CL's which are on the stack. Algorithm A is then invoked and proceeds as described in Section 3.3.1.

There are two approaches to the problem of determining whether or not a block is read-only with respect to variables which are on the PB list. The first approach, and the one currently used, is to examine the metacode for the block before executing it. A second, more sophisticated approach is to have the translator include at the head of each block a list of all the variables which are assigned values within the block.

## 8.3 File Handling

The implementation of the file handling mechanism in the DPL system gives clear evidence of the instructional intent of the processor. The tradeoff between generality and instructional efficiency has been made entirely in favor of the latter. Files in DPL can contain a maximum of 500 records. For instructional applications such a limited file size is acceptable and desirable. This number of records is large enough both to preclude manual processing of the file and to illustrate most of the variations in data among records, yet is small enough to allow an extremely efficient and controlled implementation.

When a file is OPENed during execution, a File Control Block (FCB) is created. Figure 9 illustrates the layout of an FCB, which is pointed to by the file reference variable. When a record is desired, the location is fetched from the FCB and the record is either read in from disc or used as is in core. The FCB is updated when a record is read in from disc so that subsequent references to the same record will not need a disc access. The FCB also allows direct access file organization to be maintained without reading in any information from disc as the key values are also in core. When the file is closed, all records remaining in core are written on disc and the FCB is written as a system header record.

Also written with a file are pertinent parts of the symbol table and file and record description blocks. If a program opens a file with the USING option, the symbol table stored with the file is compared with the description which the new program has used. If the two do not match, the program is denied use of the file. If the two descriptions match, use of the file is allowed, any file tags are added to the PB list, and the FCB information is read in from disc.

| |
|---|
| POSITION POINTER |
| FDB POINTER |
| CONTROL INFORMATION |
| 500 LINES, EACH CONTAINING<br><br>RECORD TYPE, LENGTH, LOCATION<br>KEY VALUE |

Figure 9. File Control Block (FCB).

## 5. SUMMARY AND DISCUSSION

DPL has been designed with two aims in mind; teaching the basic concepts needed to understand contemporary data processing systems, and testing a new programming and organizational structure for online and management information systems. The implications of the first aim as it relates to courses and curricula for contemporary data processing are discussed in Section 5.1. Section 5.3 discusses the first aim as it relates to the teaching of allied data processing subjects such as systems programming, information retrieval, and simulation.

Fulfillment of the second aim implies the use of this new organization is a real production environment. Section 5.3 describes the generalizations and extensions needed to fit the new structure into existing production languages.

5.1 Instruction in Contemporary Data Processing Systems

DPL aims to teach the non-professional user the concepts needed to understand present and future data processing systems. In particular, the clarity with which management information system concepts are expressed helps nonprogrammers to appreciate the complexity of such systems while giving them a means of classifying the parts of these systems. There are some aspects of data processing, however, which cannot be illustrated through the use of a programming language alone.

Students should certainly be given an appreciation of the importance of human engineering factors in the design of contemporary data processing systems. Forms design, production control, and the physical operation of the system should be discussed. The design of specialized input/output devices, such as those used in the online ticket reservation systems, is an example of human engineering which helps to reduce the frequency of errors made by the users of the system. Extensions to the format control ability of DPL might help in teaching forms design, and the actual use of terminals would be effective in giving the student the "hands on" experience necessary to understand some of the physical problems of system operation. It is not really practical or economical, however, to allocate either the time of the professor or the facilities necessary for students to get experience in some of the physical problems of file control, e.g., tape libraries, disc mounting, backup, etc.

Another human engineering factor which is important in management information systems design is the relevance of the information produced. The techniques of exception reporting and demand reports should be covered in an introductory course on data processing. Both of these techniques are easily demonstrated through the use of the interrupt block structure and file tagging.

A knowledge of the costs and benefits of a contemporary system is essential to a future manager who may have to decide whether or not to install a system, or which type of system to obtain. Some of these cost questions relate to the physical characteristics of the system such as physical file residence, file activity, device transfer rates, etc., all of which can affect the speed and hence the cost of obtaining certain information from the system.

There has also been some recent work on the costs and value of the information itself [17,40,57]. This type of analysis cannot really be discussed in the context of programming languages, but might be more meaningful if started after the students have gained some experience and knowledge of the large manpower effort required to write a large, integrated information system (similar to the proposal in Section 3.4).

The DPL compiler was designed with time-shared operation in mind, and will be used in this manner as soon as possible. Since the language emphasizes the use of remote terminals as input/output devices, the advantages of user interaction with such terminals should be apparent. In order to implement the time sharing features which may be called by the DPL programmer, some form of WAIT statement will be required in the language. This statement would inform the system that the programmer has no processing to do and is waiting for one of the terminals to generate an interrupt. This would aid in writing supervisor programs for terminals (as discussed in Section 3.2.)

## 5.2 Generalizations and Extensions

The file tagging and interrupt block structures of DPL make clear the interconnections between the programs and the data base of an integrated information system. Some generalizations and extensions to the DPL structures would prove extremely helpful in incorporating these features into a real production language.

PL/I [47] is a third generation programming language being given major support by IBM. Let us see, then, what additions would be needed to include the interrupt block capabilities of DPL in the PL/I language. The interrupt blocks should be treated as PL/I procedures, with the standard parameter transfer and dummy variable usage of PL/I allowed and encouraged. This eliminates the need for imposing strict coding and naming conventions when linking together the work of many programmers. The PL/I "ON condition" statement [47, p. 79], a direct analogue of the DPL PERFORM...WHEN statement, should be used as the interrupt scheduling statement. The hardware monitoring already allowed in PL/I should remain in the language and is compatible with the DPL interrupt structure.

The PB list should be treated as a named data list which can be read and possibly altered by problem programs. This would allow programmers to test for some of the conflict situations which arise by examining the PB list before adding their own conditions to it. The PB list could be treated as a character string and the string manipulation facilities of PL/I could be used to search and maintain the list.

Adding the file tagging concepts to COBOL, the programming language which is now supported by most computer manufacturers as a result of some prodding by the U.S. Department of Defense, would be a more difficult task than modifying PL/I would be. The limitations imposed on the number of records in a DPL file was a matter of instructional efficiency and could be removed without changing the structure. There is a need for a description of the structure of a file to be stored with a file itself, but this has already been proposed in a recent CODASYL report [49]. The importance of symbolic names as used in DPL would be lost in a transition to COBOL, and the relative position of a record within a file, or a variable within a record, would be used when evaluating conditions on the PB list, or when adding tags to a file.

There remains the question of whether or not a system organization as proposed in Section 3.4 can be made operationally efficient in a real production context. There are several ways in which efficiencies could be gained. First, part or all of the interrupt monitor and the interrupt block scheduler could be implemented in microprogram logic. This is well within the realm of possibility since several manufacturers are marketing systems which the users can microprogram and such systems are indicative of the trend in computer hardware [30].

Second, a more efficient systems operation could be gained by combining the interrupt block organization with a Data Processing Spooling System of the type proposed by Blomgren [8]. In this type of system, transactions would be gathered by the system throughout a specified time interval (e.g., day, week, or hour). Whenever the system deemed it necessary, the master file would be opened and the system would present the processing programs with transaction records. At this point, the interrupt block structure could take over, reading in the file tags, which would include all of the programs needed to process that file and the conditions under which each of the programs should be called. Blomgren's DPSS is not directly suited to real time updating systems, but could be used for those files which change more slowly or where the benefits of online access do not justify its cost.

## 5.3 Discussion

Modern information retrieval systems, such as SMART [51], are planning online access to a large file. The DPL system can provide a valuable tool in teaching the organization of these systems. The problems associated with organizing large files, accessing large files from remote terminals, and the techniques associated with document retrieval such as clustering can all be demonstrated with DPL. All of the instructional advantages mentioned in Chapter 1, such as simple syntax and error correction, are as useful in teaching the fundamental concepts of information retrieval as they are in teaching business data processing.

In the area of systems programming, there is as yet no language which has proven useful in teaching the relevant concepts of data management, terminal management, and program scheduling. The structures used in Chapter 3 for programming a management information system supervisor are not very different from those used to program an operating system supervisor, and the latter might be more easily learned in the controlled environment provided by DPL than in the error prone assembly level languages usually used.

The requirements of data processing languages are more closely related to those of simulation languages than is generally realized. First, the data structures used in simulation parallel in some ways those used in data processing [12,24,39]. Records in the data processing context are groups of variables, while entities in the simulation context are groups of attributes. In both cases, entities and records are structural constructs, while variables and attributes contain values. The list structures used in simulation are analogous to the use of files in data processing. A ranked list and a direct access file are both lists of records (or entities) ordered on some key value (or attribute). The use of sequential files as LIFO and FIFO lists is not quite as clean an analogy, but the pattern is clear.

The interrupt block scheduling functions used in DPL also seem analogous to the event scheduling routines used in simulation. Indeed, the OPS-3 system [25] uses a statement syntax which is similar to the PERFORM...WHEN syntax. Here, however, the analogy breaks down. In simulation, all of the events are prescheduled on the list on the basis of an internal calendar time. All blocks which are placed on a pending list have a time attribute, and will be executed when that simulated time value is reached, (which will always occur if the simulation is run long enough.) The DPL interrupt block structure does not preschedule events, and does not use any one variable to determine when execution should take place. In fact, many of the blocks on the DPL pending list will never be executed unless a particular combination of values occurs in some record.

5.4 Summary and Conclusions

A language for instruction in contemporary data processing concepts is proposed. The concepts which should be taught are spelled out in Chapter 1, along with the requirements for an instructional language. The language specification is presented in Chapters 2 and 3 and in Appendix A. The DPL system meets the requirements which are laid out.

In addition, a new organization for programming large shared data base systems is proposed and is included in the language. This method uses techniques called interrupt block scheduling and file tagging to tie together program modules and the relations of these modules to the data base. The value of this new organizational scheme is shown, both for teaching about shared data base systems and for programming such systems.

In summary, DPL should prove to be an effective tool for conveying an appreciation of the problems and promises of data processing without the requirement that every student eventually become an experienced data processing professional.

# 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 (see Figure 2, page 18).

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 characters), or reference (pointer to record, file, or device). This form is fixed for the life of the program in the Data Description Section.

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 different subscript values, each of which can be either numeric or alphabetic, e.g., PROCESS('ORDENT').

**Arithmetic Operators**

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

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

**Spacing**

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

**Functions**

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, but gives 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).

**Relations**

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

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

### Statements

The following symbols are used in the statement descriptions:

v1,v2, ... variables (text or numeric)

fr1, fr2, ... reference variables pointing to files

rr1, rr2, ... reference variables pointing to records

c1, c2, ... relational operators

slabel1, slabel2, ... statement labels

blabel1, blabel2, ... block labels

e1, e2, ... 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 ofthe 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 column 15. Columns 73 through 80 are not used by the system and may be used for identification purposes.

### Assignment Statement

| | |
|---|---|
| LET v1 = e1 | The value of the expression is assigned to the variable on the lefthand side. If v1 is a <u>text</u> variable, e1 must be either a <u>text</u> constant or a single text variable. |

**Sequence Control Statements**

GO TO slabel1

GO TO blabel1 END  Used only inside block 'blabel1'.
Causes skip to end of block.

IF e1 c1 e2 THEN s1 else s2

where s1 and s2 are any type of statement except IF. Either the THEN or ELSE phrase, but not both, may be omitted. Compound conditions of the form:

e1 c1 e2 AND e3 c2 e4 AND ... or

e1 c1 e2 OR e3 c2 e4 OR ... are allowed

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 "blabel1 BLOCK " and followed by "blabel1 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 blabel1     Executes the block once.

PERFORM blabel1 e1 TIMES where e1 has integer value.

PERFORM blabel1 WHILE e1 c1 e2 Executes the block while the condition is true. Compound conditions may be used with same restrictions as in IF.

PERFORM blabel1 FOR v1 = e1, e2, e3, ...

FOR v1 = e1 TO e2 BY e3

PERFORM blabel1 WHEN e1 c1 e2 The pair (blabel, e1 c1 e2) is placed on the pending block list. When the condition becomes true, an interrupt will be generated and block blabel1 will be executed once.

CANCEL blabel1 WHEN e1 c1 e2 The pair (blabel1, e1 c1 e2) is removed from the pending block list.

Storage Management Statements

CREATE rectype REF rr1 This allocates space for a record of type rectype, and places the pointer in rr1.

DESTROY rr1 Frees the storage associated with rr1 and sets rr1 to NULL.

Communication Statements

OPEN filtype REF fr1 Creates an instance of the file or

OPEN devtype REF fr1 device type specified, and points to it with fr1. If fr1 had previously pointed to a file of this type, the file is reopened.

CLOSE fr1 Closes the file or device pointed to by fr1.

PROTECT fr1 Causes the specified file to be made read-only.

Any write references to it will be considered errors.

POSITION fr1 AHEAD e1 RECORDS The pointer associated with
BACK e1 RECORDS the next record in the file
AT HEADER is moved to point to a different record, as directed.

READ v1, v2, ... Variables are read from the input unit.

READ rr1 FROM fr1 Next record from fr1 is read into rr1

READ rr1 WITH e1 FROM fr1 The record from Direct Access file fr1 with key=e1 is read into rr1.

WRITE rr1 IN fr1 Writes the specified record into file fr1. Position in the file is determined by key if direct access, else record is added to end of the file.

<u>Comments</u>

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

APPENDIX B

## Details of Implementation

### Introduction

The reader is assumed to be familiar with the contents of Chapter 4, including Figures 4 through 9. This Appendix will go into some detail about the subroutine structure of the DPL compiler and execution monitor, but will avoid the bit-chasing details which are apt to change as bugs are removed from the system. Anyone interested in the latest version of the DPL compiler, or the up to the minute status of a particular routine is invited to write the author at the Department of Computer Science, Cornell University, Ithaca, New York, 14850.

All of the programs and interfaces described below are coded in 360 Assembler Language, and communicate with one another through the use of branch tables. This language was chosen to be compatible with the CUPL compiler implementation. Many of the subroutines, and the overall structure have been copied from that implementation, details of which are also available from the author.

### Control Program

The control program is responsible for operating system interfaces, user interfaces, maintaining flow of control, and providing common services. The routines are listed below:

GOSTART main entry point, DPL initialization
GOREADCD reads a card
GOPRINTL prints a line
CVCSTOFP converts character string to floating point
CVFPTOCS converts floating point number to character string
GOGETCOR allocates a block of storage
GOFRECOR frees a block of storage
GOGETDS retrieves a dataset
GOPUTDS saves a data set
GOSCNSET sets up registers for translator
GOINTSET sets up registers for interpreter
GOPRERR formats and prints error messages
GOSETTIM sets time and other job limits
GOPCEXIT takes program check interrupts
GOENDIT terminates user program normally
GOABORT terminates user program abnormally
GOBOMB terminates compiler with dump
GODUMPST dumps symbol table and metacode for debugging

The flow of control is as follows: The operating system passes control to GOSTART, which initializes the compiler. This also reads in the student's ID information and the limits on time, pages, etc., for his job. Control is then passed to GOSCNSET, which sets up for the translator to scan the Data Description Section. The translator returns control to GOSETINT, which sets up for the interpreter and passes control to the interpreter. When the job is completed, the interpreter passes control to GOAGAIN, which looks for the next student job, and passes control back to GOSTART.

Translator Program

The translator program actually operates in three distinct phases. The first, SCAN, scans the Data Description Section and builds the symbol and constant tables, and associated record and file description blocks. When this is completed, control is passed to SDRI, which scans the program statements and generates metacode, and then control is given to SCLEANUP which places the final touches on the metacode.

CARD is the lexical analysis routine called by the other translate time routines. It determines which characters on the card form a valid DPL construct, searches the appropriate tables, and returns the metacode symbol for that construct. In addition, CARD contains the symbol and constant table management routines, and is responsible for searching and inserting into these tables.

SCAN reads the Data Description Section cards, places the information into the symbol table and record description blocks, and allocates storage for arrays which are not part of records. Simple variables which are not mentioned in the

Data Description Section can be declared implicitly by their appearance in the source program.

SDRI has a subroutine for each statement type, in which the actual syntax of that statement is checked. When an error is discovered, the SERROR routines are called to place an error message number in the error stack. At the end of each statement, the error stack is checked, and, if a severe enough error has been detected while scanning this statement, REVERSE is called. This program will produce a source image from the metacode which has been generated for the statement.

The SDRI statement drivers use the VARI subroutines to check the syntax of variables and arithmetic expressions. These VARI routines use a standard recursive single stack scan for expressions, with a minor modification to handle the pointer notation of DPL.

At the end of the first pass through the source program, the SCLEANUP routine is called. This subroutine handles those conditions which cannot be detected in the single pass scan, e.g., missing labels, undefined variables, etc. A spelling correction routine is called and tries to determine whether or not some of the suspiciously used variables or labels are actually misspellings of other, properly used variables and labels. If they are, the symbol table entries are altered so that all references to the misspelling are directed to the properly spelled occurrence. The SCLEANUP routine returns control to GOSETINT which starts the interpreter program.

<u>Interpreter Program</u>

The interpreter program has three main components; the main control or execute section, the interrupt scheduler and monitor, and the file handler. The execute section directs the actual interpretation of the metacode, much as the 360 microprogram interpreter works. This section uses the statement type as an operation code, and transfers to an appropriate routine for that statement type. These routines may in turn call for memory accesses, which in DPL involve either the symbol table or record storage. A set of routines to interpret arithmetic expressions and addresses are provided, and are coded in a manner similar to the VARI routines of the translator.

Whenever any of the statement routines wishes to perform a store operation, the interrupt monitor is called. This routine checks to see whether or not the variable which is being assigned a value is involved in some condition which is on the PB list, and goes through the checking indicated in Chapter 3.

PERFORM...WHEN statements call the interrupt scheduler, which actually maintains the PB list as shown in Chapter 3. This routine is also called by main control if the interrupt monitor has generated any interrupts during the execution of the previous DPL statement.

The file handler aids the OPEN, CLOSE, READ and WRITE statement drivers by handling all accesses to files. This modularity aids in the system to system transition discussed in Chapter 4. The file handler is in communication with the GOGETDS and GOPUTDS routines when physical file I/O is required.

When the interpreter executes a STOP statement, some final diagnostic information is provided to the user, e.g., final values of all simple variables, the number of times each label was entered, and the statement numbers of the last 15 statements executed. Control is then returned to GOENDIT, which in turn calls GOAGAIN for the next user.

Intermediate Language

In the following discussion, a byte refers to the 360 8 bit byte, which can contain two hexadecimal (base 16) digits. These digits are numbered 0 to 9 and A to F.

In metacode, symbols representing entries in the symbol or constant tables require two bytes, while all other symbols require one byte. In the following syntax, each byte is represented by the pair mn, where m is the upper half byte and n is the lower half byte digit. A symbol table line number will be denoted stnn, and a constant table line number will be denoted ctnn. Other symbols are as defined below.

Variables denoted vvvv.

a. not in records

simple 01 stnn

vector 01 stnn A9 eeee AA

array 01 stnn A9 eeee B0 eeee AA

b. in records

09 vvvv B1 elem

Elements of a record denoted elem.

simple 08 stnn

vector 08 stnn A9 eeee AA

array 08 stnn A9 eeee B0 eeee AA

Constants denoted cccc

02 ctnn

Arithmetic Expressions denoted eeee

vvvv and cccc combined with arithmetic operators and functions.

Write elements denoted wrtel

vvvv for variable

D6 vvvv for /v

DC vvvv for $v

03 ctnn for messages

Labels denoted as llll

simple 04 stnn

vector 05 stnn nnnn

Boolean Expressions denoted bbbb

eeee B8-BD eeee (simple relation)

bbbb D9-DA bbbb (compound relation)

start of statement denoted start

F0 line number

FO line number 1111

end of statement denoted endst

F4

Unlabeled statement denoted ssss

Reference variables recrf, filrf, devrf all instances of

vvvv

statements of DPL

1. LET Statement

CO vvvv A0 eeee

2. GO TO Statement

C1 1111

3. IF Statement

C2 bbbb F1 line ssss (IF..THEN form)

C2 bbbb F1 line CB F2 line ssss (IF..ELSE form)

C2 bbbb f1 line ssss F2 line ssss (IF..THEN..ELSE form)

4. READ Statement

C3 special byte followed by

vvvv DA ... D8 vvvv dd devrf

recrf DD devrf

recrf DD filrf

recrf DE eeee DD filrf

the special byte tells whether or not this is a file or a device directed read, and points to the filrf or the devrf in the metacode.

5. WRITE Statement

C4 special byte as in READ followed by:

wrtel D8 ... D8 wrtel DD devrf (variable list form)

recrf DD devrf (write record on device)

recrf DD filrf (write record in file)

d7 (write ALL)

6. PERFORM Statement

C5 llll followed by

D0 eeee (TIMES)

D1 bbbb (WHILE)

DB bbbb (WHEN)

DB devrf (WHEN devrf INTERRUPTS)

D2 vvvv eeee D3 eeee D4 eeee (FOR.TO.BY)

D2 vvvv eeee D8 ... D8 eeee (FOR list)

7. STOP statement

C6

8. No-op

CB

9. CREATE statement

E0 01 stnn recrf (stnn points to record type)

10. DESTROY statement

E1 recrf

11. OPEN Statement

E2 01 stnn filrf [ EE A(DSCB) ]

stnn points to file type, DSCB is for USING option.

12. CLOSE Statement

E3 filrf [ ED A(DSCB) ]

bracketed expression is for SAVE option.

13. CANCEL Statement

E5 1111 DB bbbb (WHEN condition)

DB devrf (WHEN device INTERRUPTS)

14. PROTECT statement

E6 filrf

15. POSITION statement

E4 filrf 82 eeee (AHEAD eee RECORDS)

83 eeee (BACK eeee RECORDS)

84 (AT HEADER)

16. BLOCK statement

C9 1111

17. END Statement

CA 1111

Symbol Table and Constant Table

The symbol table can contain up to 225 entries, which are hashed into the table on the basis of the first three letters of the identifier. Each entry occupies 24 bytes. The first 8 bytes contain the EBCDIC representation of the name, and the ninth byte tells what type of entry this line is.

The tenth byte is used to chain the symbols in the table in the order in which they were entered into the table. This is used for diagnostic purposes. The remaining bytes are different for each different type of entry, and are described below:

<u>TYPE Usage of Other Bytes</u>

VS Simple variable.

11 form and usage

14-16 A(CL)

17-24 value

VA

VV Vector or array variable

11 form and usage

14-16 A(SL)

17-18 first subscript limit

19-20 second subscript limit

21-24 A(array)

RV

RA

RS simple, vector, or array record element

11 usage

12 record type belonged to

13-16 displacement within the record

17-18 first subscript limit, if applicable

19-20 second subscript limit, if applicable

LS

BS simple statement or block label

11 defined and referenced flags

13-16 location of the statement in metacode

17-20 number of times this label was entered

LV

BV vector statement and block labels

11 defined and referenced flags

13-16 A( SLCB) subscripted label control block

TF file type

13-16 A( FDB) file description block

TD device type

12 number of devices allowed

13-16 A(DCB) device control block

17 input/output/interrupt switches

TR record type

11 number of variables in this record

13-16 A(RDB) record description block

17-20 size of record in double words

The record and file description blocks (FDB's and RDB's) are simply lists of symbol table line numbers for the record elements or the record types which are allowed in the particular record or file type. The SLCB, or subscripted label control block has on entry for each subscript value which is allowed. This entry is eight bytes long and is used as follows:

1 line number in constant table for this subscript value

2 definition and usage flag

3-5 count of number of times this label is used

6-8 location of the statement in metacode

Values in DPL are eight bytes long. A floating point number uses these bytes for a double word 360 floating point number. Text variables store eight characters of text.

Reference variables use the bytes as follows:

1 stnn of record, file, or device pointed to.

2-4 address of SL for thing pointed to

5-8 address of record, FCB or DCB pointed to.

The constant table entries are all 10 bytes long and contain both the value of the constant and information which can be used to reconstruct the character string which was read in. The first eight bytes are devoted to the value and the last two bytes are used as follows:

| byte | bits | usage |
|---|---|---|
| 1 | 0 | on if decimal point punched |
| 1 | 1 | on if exponent entered as Enn |
| 1 | 2 | on if this is text constant |
| 1 | 3-7 | give number of digits entered |
| 2 | | exponent modifier (no. of places to left of 1st digit that decimal point was placed) |

## REFERENCES


1. Arden, B., B. Galler, and R. Graham, Michigan Algorithm Decoder, University of Michigan, January 1963.

2. "Associative Programming Language," Computing Technology Dept., General Motors Research Lab, Warren, Michigan.

3. Bash, J. L., E. G. Benjafield, and M. L. Goudy, The MULTICS Operating System, Cambridge Information Systems Laboratory, May 1967.

4. Bauer, C. R., A. Peluso, and W. S. Worley, Jr. IITRAN/360 - Self Instructional Manual and Text, Reading, Mass.: Addison Wesley, 1967.

5. Canning, R., EDP for Business and Industry, New York: Wiley, 1966.

6. COBOL Language Specifications, IBM Manual No. C28-6516-6, 1965.

7. COGENT II General Information Manual, Computer Sciences Corporation, 1968.

8. Conway, R. W., and G. H. Blomgren, "Preliminary Description: A Spooling System for a Data Processing Environment," Technical Report No. 68-A3, Office of Computer Services, Cornell University, Ithaca, N. Y.

9. ______, J. J. Delfausse, W. L. Maxwell, and W. E. Walker, "CLP - The Cornell List Processor," Comm. ACM 8 (April 1965), 215-216.

10. ______, and W. L. Maxwell, "CORC - The Cornell Computing Language," Comm. ACM 6 (June 1963), 317-321.

11. ______ and ______, "CUPL- An Approach to Introductory Instruction," Technical Report No. 68-2, Dept. of Computer Science, Cornell University, Ithaca, N. Y. 1968.

12. Dahl, O. J., and K. Nygaard, "SIMULA - An ALGOL Based Simulation Language," Comm. ACM 9 (Sept. 1966), 671.

13. Dearden, J., and F. W. McFarlan, Management Information Systems, Homewood, Ill.: Richard D. Irwin, 1966.

14. Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," Comm. ACM 9 (March 1966), 143-155.

15. Desmonde, W. H., Real Time Data Processing Systems: Introductory Concepts," Englewood Cliffs, N. J., Prentice Hall, 1964.

16. Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control," Comm. ACM 8 (September 1965).

17. Emery, J. and D. Ness, "A Man Machine Budgeting System." Paper presented at the Conference for Research on Management Information Systems, Carnegie Mellon University, June, 1968.

18. Farber, D. J., R. E. Griswold, and I. P. Polonsky, "The SNOBOL3 Programming Language," Bell System Tech. J. L_ , (July - Aug 1966), 895-944.

19. Feldman, J. and D. Gries, "Translator Writing Systems," Comm. ACM 11, (Feb. 1968), 77-113.

20. FORTRAN IV Language Specifications. IBM Manual No. C28-6515-5, 1966.

21. FORTRAN 63 Reference Manual. Control Data Corporation Pub. No. 60174900, Oct., 1966.

22. Freeman, D. N., "Error Correction in CORC: The Cornell Computing Language." Ph.d. Thesis, Cornell University, Ithaca, New York, Sept. 1963.

23. GIS. IBM 360 Applications Program Announcement, 1968.

24. GPSS/III: Introduction. IBM Manual No. B20-0001-0,1966.

25. Greenberger, M., M. Jones, J. Morris, and D. Ness, On-line Computation and Simulation: OPS-3. Cambridge Mass: MIT Press, 1965.

26. Gregory, R. and R. L. Van Horn, Business Data Processing and Programming. Belmont, Calif.: Wadsworth Pub. Co., 1963.

27. ______, and ______, Automatic Data Processing Systems: Principles and Procedures. 2nd ed. Belmont, Calif.: Wadsworth Pub. Co., 1963.

28. Head, R. Real Time Business Systems. New York: Holt, Rhinehart, and Winston, 1964.

29. Hendry, D. Provisional BCL Manual. Institute of Computer Sciences, London, England, 1966.

30. "IBM Follows a Tiny Leader," Datamation 14 (Aug. 1968), 89.

31. "Integrated Data Store," Information Systems Marketing, General Electric Company, 1967.

32. IBM 709 Data Processing System. IBM Manual No. A22-6536, 1959.

33. IBM 7040 Principles of Operation. IBM Manual No. A22-6649, 1962.

34. Iverson, K. A Programming Language. New York: Wiley, 1962.

35. Jackson, M. A., "The Need For Imprecision," Datamation 14, (Feb. 1968), 143-144.

36. Kemeny, J. and T. Kurtz. BASIC Programming. New York: Wiley, 1967.

37. Lombardi, L., "A General Business Oriented Language Based on Decision Expressions," Comm. ACM 7, (Feb. 1964).

38. McCarthy, E. J., J. A. McCarthy, and D. Mines. Integrated Data Processing Systems. New York: Wiley, 1966.

39. Markowitz, M., B. Hausner, and H. Karr. SIMSCRIPT: A Simulation Programming Language. Englewood Cliffs, N. J.: Prentice Hall, 1963.

40. Marschak, J., "Efficient Choice of Information Services." Paper presented at the Conference for Research on Management Information Systems, Carnegie Mellon University, June 1968.

41. Martin, J. Programming Real Time Computer Systems. Englewood Cliffs, N. J.: Prentice Hall, 1965.

42. Moulton, P. G., and M. E. Muller, "DITRAN -- A Compiler Emphasizing Diagnostics," Comm. ACM 10, (Jan. 1967), 45-52.

43. OS/360: Concepts and Facilities. IBM Manual No. C28-6535-1, 1965.

44. OS/360: Data File Handbook. IBM Manual No. C20-1638-1, 1965.

45. Opler, A., "Fourth Generation Software," Datamation 13, (Jan. 1967), 22-24.

46. Parkhill, D. F. Challenge of the Computer Utility. Reading, Mass.: Addison Wesley, 1966.

47. PL/I Language Specifications. IBM Manual No. C28-6571-4, 1965.

48. Postley, J. A., "The MARK/IV System," Datamation 14, (Jan. 1968), 28-30.

49. "Report to the CODASYL COBOL Committee: COBOL Extensions to handle Data Bases," SIGPLAN Notices 3 (April 1968).

50. Rosen, S., R. Spurgeon, and J. K. Donnelly, "PUFFT -- The Purdue University Fast FORTRAN Translator," Comm. ACM 8, (Nov. 1965), 661-666.

51. Salton, G., "Search and Retrieval Experiments in Real-time Information Retrieval," Technical Report No. 68-8, Dept. of Computer Science, Cornell University, Ithaca, New York, 1968.

52. Saxon, J. and W. Steyer. Basic Principles of Data Processing. Englewood Cliffs, N. J.: Prentice Hall, 1967.

53. Small Computer Handbook. Digital Equipment Corporation, 1967.

54. Shantz, P., R. A. German, J. G. Mitchell, R. Shirley, and C. Zarnke, "WATFOR -- The University of Waterloo FORTRAN IV Compiler," Comm. ACM 10 (Jan. 1967), 41-44.

55. System 360 Basic Assembler Language. IBM Manual No. C28-6514-5, 1964.

56. System 360 Principles of Operation. IBM Manual No. A22-6826-6, 1967.

57. "Trouble..I Say Trouble in DP City," Datamation 14 (Aug. 1968), 21-22.

58. Walker, R. J. An Instruction Manual for CUPL. Cornell University, 1966.

59. Wegner, P. Programming Languages, Information Structures, and Machine Organization. New York: McGraw Hill Book Company, 1968.

60. Wilkinson, J. H. Rounding Errors in Algebraic Processes. Englewood Cliffs, N. J.: Prentice Hall, 1964.