The amount of time which has elapsed between Liebritz's first theoretical description of a computing machine and today's commonplace use of digital computers as extensions of Man's intellectual faculties is a little more than three hundred years. This period has witnessed the birth and death of many trends in the art of mechanical computing, some no more than fads and others becoming established as fundamental truths which are now accepted as axioms of computer science. One of the most firmly established of these latter trends is the quest for generality. As early as 1833, Charles Babbage discerned that in the ideal computing machine, the human operator should have completely flexible control not only over the data to be processed, but also over the algorithms.

The history of computing since Babbage has been marked by a terrific emphasis upon flexibility. Turing and Von Neuman laid a theoretical basis of generality in control structures. Recent work in macro-modules (Clark and Bell) has started an explosion in the generality even of hardware. On the software scene, the growth of the concept of modular "structured programming" has opened new horizons in the generality of the programming process. However, it is the myriad computer languages that most clearly point out the trend towards generality. Of the hundreds of programming languages which have grown up in the past decade, all but a few have been designed with a delineated scope of applicability. No matter how well such a language covers its particular field, it is forever relegated to that very restricted area. The most important languages of the modern era, FORTRAN, PL/1, ALGOL, and COBOL, are all marked by a common feature: each provides facilities general enough and powerful enough to implement a wide class of algorithms.

Sadly, the field of information processing and data base management has lagged behind these trends. It seems that the field of information processing systems is characterized by a welter of small systems each designed to implement a given complement of algorithms (such as storage, retrieval, and cross-tabulations). While this set of algorithms is often very large, and the designers may have anticipated many of the needs of the target installations, still we await the advent of a data base management which satisfies Babbage's ideal of flexible control.

The DUCHESS project marks an attempt to design an information management system containing facilities for implementing a wide range of data manipulation algorithms. To this end, we have taken steps to generalize the three basic subdivisions of an information system: the data base, the control structure, and the operating system which supports the system. The primary emphasis in the following discussion will lie upon the desirability and practicality of making every feature of the system as general as possible.

It would be appropriate at this point to present an overview of the DUCHESS system to serve as a reference for discussions to follow. DUCHESS is implemented on a DEC PDP-11 minicomputer with 28 K words of memory. Peripheral resources available to the system include: two RK11 disk drives (2.4 megabytes each), a console teletype, and several CRT-type conversational terminals.

The software side of the DUCHESS system may be divided into four distinct modes: (1) a data base access and management system; (2) a high level programming language with which to implement the data management algorithms. This language provides facilities for interfacing with the data base access module; (3) a complement of user service routines which implement often-used system functions; (4) a multi-user executive system which multiplexes the resources outlined above between multiple users, each in control of one of the CRT terminals. Because the length of this paper is limited, only the first two modules will be discussed. The user routines and the executive are essentially transparent to the user and do not significantly relate to the concept of generality which this paper illustrates.

DATA BASE—DATA DEFINITION LANGUAGE

The first question facing the designer of any data management system is "What data will the system need to record?" In the case of DUCHESS the answer to this question came easily; we would provide facilities for recording any type of data which can be transcribed into a computer-readable format. The next step was to select a suitable set of datatypes...
for the representation of data. The following set was chosen as adequate to record any type of data:

<table>
<thead>
<tr>
<th>DATATYPE</th>
<th>DESCRIPTION</th>
<th>INTERNAL STORAGE FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CODE (n)</td>
<td>This type is used to record a selection from within a set of n mutually exclusive choices. Typically, “multiple choice” data is recorded in this format.</td>
<td>log n contiguous bits</td>
</tr>
<tr>
<td>2. BOOLEAN</td>
<td>A special case of the CODE datatype used to record yes/no data. The only permissible values are 0 (false) and 1 (true).</td>
<td>1 bit</td>
</tr>
<tr>
<td>3. FIXED (d, p)</td>
<td>A fixed point number of d digits with p digits to the right of the implied decimal point. The default for d and p is 6 and 0 respectively.</td>
<td>1 or 2 words (single or double precision is chosen by the values of d and p). Decimal Point location is implicit.</td>
</tr>
<tr>
<td>4. FLOAT</td>
<td>A floating-point number of standard PDP-11 floating point format</td>
<td>2 words mantissa, 1 word exponent</td>
</tr>
<tr>
<td>5. DATE</td>
<td>This datatype is used to record dates (a frequently used datatype) in a compact format.</td>
<td>1 word. The date is compressed upon storage and re-expanded at retrieval.</td>
</tr>
<tr>
<td>6. TIME</td>
<td>Like the DATE datatype, this datatype records often-used sort of data in a compact format.</td>
<td>1 word. Time is expressed as hours, minutes and seconds since midnight.</td>
</tr>
<tr>
<td>7. TEXT (n)</td>
<td>This type is used to record character strings known length (as, for example, a set of three initials). Character strings are truncated or padded with blanks to fit the storage field.</td>
<td>n consecutive bytes (0&lt;n&lt;256)</td>
</tr>
<tr>
<td>8. TEXT</td>
<td>This is the general case of the TEXT datatype used to record varying length character strings, such as names and addresses with no wastage.</td>
<td>1 word pointer into a “free text buffer.” Within the free text buffer—one byte for each plus one length byte (0&lt;length&lt;256)</td>
</tr>
</tbody>
</table>

It is immediately evident that, by recognizing many different datatypes and choosing the most efficient representation for each, DUCHESS can gain a considerable savings in data storage space over other systems. Consider, for instance, a certain item of data which may take on any one of four different states. It is evident that the storage of this datum should take no more than two bits to record the alternatives. However, most data management systems allocate space for non-text nodes on word boundaries. This means that, on a 16-bit machine like the PDP-11, 14 bits are wasted. In a data base of ten thousand records, this wastage amounts to more than 8.5 K words of wasted storage space which could have been reclaimed if an appropriate data type (like CODE(n)) with allocation on bit boundaries, had been available. DUCHESS solves the problem of data alignment within records by aggregating all bit-aligned variables into one field, all byte-aligned variables into another field, and all word-aligned variables into a third field. In this way, only one “filler” space is required per field.

It was decided that DUCHESS needed facilities to record both fixed and variable length character strings. Fixed length strings are stored in the most obvious manner: the appropriate number of contiguous bytes are allocated to the string. The length byte which normally precedes a DUCHESS character string is not carried in the record because the length of the string is bound via the data definition facilities. The fact that the location of a fixed-length character string is always known as an offset provides for rapid access via simple address arithmetic. For variable-length strings, the process is somewhat more complex. One word of storage is allocated in the fixed length section of the record for a pointer into free text buffers which are allocated on demand. Thus, it takes two references to access a variable-length text string stored in a DUCHESS record.

The most natural format devised for a data base representation is that of a multi-level hierarchy. It speaks well of the information management field that this format has been almost universally adopted. Human beings tend to look at a large mass of data (like that represented in a typical data base) in terms of its logical subdivisions. Hence, the structural hierarchy fits very naturally into a human’s conception of the information recorded in a data base. DUCHESS, like other hierarchical information systems, faces a significant problem in converting a human’s data base definition specifications (called the “hierarchy document”) into a compact model of the format of a data record.

The user must describe the data to be recorded by preparing a “hierarchy document” for the system. The format of this document is very similar to the format of a PL/1 structure. As an example, we will consider the following
portion of a hierarchy:

1 BOOK SEGMENT
2 AUTHOR SEGMENT
3 NAME
4 FIRST TEXT
4 LAST TEXT
4 INITIAL TEXT (1)
2 REVIEWER SEGMENT
3 NAME
4 FIRST TEXT
4 LAST TEXT
4 INITIAL TEXT (1)

This document is used as input into a "hierarchy compiler" which constructs a model of the record format for the data base. This compiled version is used to create and initialize the data base file and in compiling application programs.

The above example illustrates two forms of data partitioning under DUCHESS. The simplest form is illustrated by the level 3 node called NAME. Note that this node serves purely as a structuring device and carries no data whatsoever. However, one bit is allocated for each such node in the record. This bit is initially clear and is set whenever one of the nodes below it is given a value. This node type acts as a BOOLEAN node, so that it is possible to poll a node such as NAME and determine whether or not anyone of its sub-items has been altered from its initial state. The assignment of a value to a variable automatically invokes the mechanisms to post such nodes, a process called upward spreading. Of course, this spreading action will continue upward as far as necessary, each spreading operation triggering a spread to the next higher level until a previously posted level or the root node is reached.

The above example also illustrates the use of the segmented data structure in the DUCHESS data base. This concept of a data-dependent data structure is one of the most powerful features of the DUCHESS system and deserves somewhat of an introduction. One of the most significant problems facing a user of a data base system is the tradeoff between complete freedom in hierarchy planning and the advantages of associating fixed offsets with node information. To this end, there is another DUCHESS datatype not mentioned in the preceding list, the SEGMENT type. To the person generating the hierarchy, the use of the SEGMENT datatype signifies that all nodes immediately subordinate to the segment node are considered as a single logical entity. In terms of our example, assume for a moment that the REVIEWER substructure is not used in every record. By coding REVIEWER with the datatype SEGMENT, the developer indicates that space is not to be allocated in the record for the REVIEWER substructure until one of the terminal sub-fields (FIRST, LAST, or INITIAL) are assigned a value. When the value of any of these fields is recorded in the record, an appropriate amount of space is allocated for the entire segment containing the target node. The allocated space is then linked to its "parent" segment in the record so that later access to the data can descend from the father into the newly allocated segment. It may be useful here to present a visual comparison between the "standard" storage technique and the DUCHESS segment structure. In our example, assume that the only information recorded is the name of the author, JOHN Q. SMITH. The REVIEWER substructure is still in the initial state. Compare the two storage structures diagrammed in Figure 1.

Data segmentation is an attempt to strike a workable balance between complete freedom in hierarchy planning and the advantages of associating fixed offsets with node information. To this end, there is another DUCHESS datatype not mentioned in the preceding list, the SEGMENT type. To the person generating the hierarchy, the use of the SEGMENT datatype signifies that all nodes immediately subordinate to the segment node are considered as a single logical entity. In terms of our example, assume for a moment that the REVIEWER substructure is not used in every record. By coding REVIEWER with the datatype SEGMENT, the developer indicates that space is not to be allocated in the record for the REVIEWER substructure until one of the terminal sub-fields (FIRST, LAST, or INITIAL) are assigned a value. When the value of any of these fields is recorded in the record, an appropriate amount of space is allocated for the entire segment containing the target node. The allocated space is then linked to its "parent" segment in the record so that later access to the data can descend from the father into the newly allocated segment. It may be useful here to present a visual comparison between the "standard" storage technique and the DUCHESS segment structure. In our example, assume that the only information recorded is the name of the author, JOHN Q. SMITH. The REVIEWER substructure is still in the initial state. Compare the two storage structures diagrammed in Figure 1. For purposes of illustration, the structures shown above have been considerably simplified. Nonetheless, the substantial savings in unused space within the record should be im-

![Figure 1 - Fixed vs. segmented record format](image-url)
mediately evident. The DUCHESS structure uses one word for a null pointer to represent the missing REVIEWER structure (always with the option of filling in the structure by allocating space for the segment and establishing the REVIEWER link in the BOOK segment). By contrast, the fixed record format requires many bytes of unused space, simply to maintain the record structure. Thus, an infrequently used item costs the developer considerably less if he uses the DUCHESS system of allocation. We feel that this fact contributes significantly to the flexibility of the system.

Because the tradeoff of data to be recorded versus storage used is so important to the data base developer, DUCHESS provides a facility similar to the sequential search method mentioned earlier. Each segment of the hierarchy may contain certain nodes labeled SPARSE. Such nodes generally represent data which will only rarely be stored in the record. Associated with each segment is a variable sized “sparse buffer” to contain the values of all the SPARSE nodes declared within the segment. When none of the SPARSE nodes is assigned, the sparse buffer is not allocated and takes up no storage. Upon assignment of a SPARSE node, a sparse buffer is allocated within the record and linked to the parent segment. Within this sparse buffer, the data of the SPARSE node is recorded, preceded by a node identifier. At retrieval time, the sparse buffer is sequentially searched for the node identifier requested. This process seems identical with the sequential technique mentioned earlier. However, since SEGMENT nodes can be declared as SPARSE, the DUCHESS implementation of this data structure actually represents a mixture of fixed and variable record structure.

At this point, we will anticipate some objections by admitting that the preceding paragraphs glossed over some serious problems raised by the new structure. In a fixed record format system, each datum has a certain offset from the beginning of the record and is instantly accessible by a simple address calculation. In the DUCHESS segmentation system, each item of data is associated with an offset from the base of its segment. The process of locating the segment is mainly one of following a linked list through the data base file. We admit that this is a potentially horrendous task. However, we have succeeded in building enough “intelligence” into the system so that nearly all of the list searching can be obviated and the remaining traversal will not seriously detract from system performance. Once again, we feel that the flexibility gained by implementing this more sophisticated record structure is well worth the extra processing involved. The technique of avoiding the inherent segment lookup is discussed in the next section on the DUCHESS programming language.

**DUCHESS PROGRAMMING LANGUAGE**

We have already described in part our effort to produce a powerful and flexible data base structure. We believe that this data structure can efficiently record any sort of data which can be transcribed into the computer’s input devices. However, this structure would be virtually worthless without an equal degree of freedom in the facilities available for manipulating the recorded data. As mentioned earlier, most data management systems provide a stock set of algorithms for data manipulation. However extensive this set may be, it cannot possibly cover all the needs of an installation which makes frequent use of a data base. A decision was made early in the DUCHESS development to implement a high-level language with two central features: (1) a set of commands designed specifically to interact naturally with the DUCHESS data base system; and (2) an instruction set of sufficient generality that any data-processing algorithm could be realized without having to “strain” either the language or the data base. It order to stay close to a familiar model, we decided to pattern our language after PL/1. The primary difference between the DUCHESS language and PL/1 lies in three areas: (1) the DUCHESS language is a subset of PL/1; we saw no need to implement the more esoteric features of full PL/1. (2) A full set of CRT terminal input/output commands has been added to the GET/PUT repertoire to take advantage of the conversational abilities of a video terminal. (3) The variable reference structure of PL/1 has been supplanted by one which interfaces more naturally with the DUCHESS data base structure. It is this third factor which is discussed in the remainder of the paper.

Recall the objections to the operational characteristc of the DUCHESS segmented data structure raised in the last chapter: the process of looking up a segment through the segment pointers is a slow operation and costly in storage accesses. If a “segment lookup” was required for each reference to a node in the data base, the system would be impossibly slow. However, it is only necessary to perform the “segment lookup” on the initial reference to a segment. The use of a concrete example might serve to clarify the process of “descending” through the segments. Consider the process of setting the name of the author to JOHN Q. SMITH. The requisite DUCHESS statements are:

```
AUTHOR.NAME.FIRST = 'JOHN';
AUTHOR.NAME.LAST = 'SMITH';
AUTHOR.NAME.INITIAL = 'Q'
```

The actions performed are as follows:

1. Look up the BOOK segment
2. Look up AUTHOR segment within the BOOK segment
3. Assign NAME.FIRST
4. Assign NAME.LAST
5. Assign NAME.MIDDLE

Notice that only two “segment lookup” operations are necessary to locate the segment AUTHOR. Once this segment is located, all operations within the segment become automatic since the segment base is known. To highlight another feature of this process, consider the job of setting the name of the reviewer to JAMES H. DOE after having carried out the process above:

1. Discard the base of the AUTHOR segment
Because the segment BOOK had been located in the process preceding this one, the base address of the BOOK segment was already known and hence it was unnecessary to search for it again.

This process of remembering the locations of segments in the "active reference path" is known as "locality management." A locality is the address within the file of a given segment of a given record and the addresses within the record of all its "father" segments. A locality behaves much like a stack: when a program references a segment "deeper" in the hierarchy then the top reference, the top segment of the locality is used as a starting site for the segment lookup, thus short-circuiting part of the lookup procedure by using data already on the stack. Similarly, when a reference is made to a node "higher" in the hierarchy or down a different branch of the hierarchy tree, segments are "popped" off the locality stack until either the target segment is the top one on the locality stack or until the locality stack is in an appropriate state to begin the lookup procedure (as in the preceding example). In order not to overwork a single locality stack, DUCHESS can maintain up to 256 separate active localities. Thus if an application program needs to reference data from several different segments concurrently, each of the segments can be included in the set of active localities. In these ways, proper locality management in the DUCHESS language removes the primary objection to use of the segmented data structures.

We have described a system of locality management that lets the programmer move around efficiently within the hierarchy structure. However, we find that this new freedom imposes tremendous responsibilities upon the programmer to set up the localities properly before attempting to access a node of the file. Clearly, this new expansion of responsibilities is unacceptable. To confront this new problem, we introduced an element of "intelligence" into the system by placing the bulk of the responsibility of locality management upon the DUCHESS compiler. The compiler accepts as input not only the user's program but also the processed versions of the hierarchy documents for all files to be accessed in the program. Through use of these documents, the compiler has an intimate knowledge of the structure of the data base. When the compiler is called upon to generate code to access a node of the data base, it can check the set of current localities open (even at compile time) for a locality which will provide the appropriate segment base. If no such locality is available, code is generated to establish a locality stack for such use. Although the user retains the facilities for explicit locality control (and he may exercise explicit control over key localities in order to "fine tune" system operation), the compiler has the facilities for carrying out all the "dirty work" of locality management.

In order to ensure that the compiler has enough information to properly manage the localities, it is important that the user write code which follows the stack nature of locality growth. That is, any references to a node in the data base should be preceded by the appropriate locality management code (generated either by the compiler or by the user). At compile time, it must be possible to exhaustively enumerate the paths the program control might take and, in the code for each path, generate the appropriate locality management code. The major obstacle to such analysis is the abuse of the GOTO statement. The target statement of a GOTO has two distinct entry paths, one by natural program flow and the other by the action of the GOTO statement. Each of these entry locations requires a different sort of locality management to bring the active localities into the state needed by the target statement. In the case where one statement is the target of multiple GOTO statements, the problem becomes, for all intents and purposes, insoluble. Apparently, we need a mechanism for insuring that each DUCHESS statement has exactly one entry path: the natural program flow. The mechanism chosen is distressingly simple; the DUCHESS compiler prohibits any form of the GOTO statement. To assume the function of this control statement, a set of control structures have been implemented, providing such facilities as bypassing a group of instructions, iterative looping, looping on condition, and a form of multi-way branching. The basis of these control structures results from current investigations in 'structured programming.' It has been shown that not only can the task of compilation be made more efficient and object code be optimized but surprisingly, programs written with such restraints are easier to debug. We anticipate that new programmers, after an initial period of adjustment, will adapt well to a new sort of control structure.

In this paper (which, in retrospect, seems woefully inadequate to convey the new ideas generated in the course of the DUCHESS design) we have tried to illustrate that the primary problem with most modern information systems lies in an inherent rigidity both in their data structure and in the control structure. This rigidity tends to lead their users into assuming that the scope of information management systems should include only those functions presently implemented: storing, retrieval, searching and sorting, and elementary analysis. The ideal information system should have these capabilities plus opportunities for developing individualized procedures, via a programming language or other means. We hope that the DUCHESS project will illustrate that the practical implementation of such a flexible system is not only possible but no more difficult than the design and implementation of a rigidly restricted system. Furthermore, we demonstrate that this flexibility does not come with a higher price tag, for DUCHESS efficiency compares favorably with its less flexible predecessors.

REFERENCES


