A SPECIAL-PURPOSE LIST PROCESSOR

by

BURTON JORDAN SMITH

B.S.E.E., University of New Mexico
(1967)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1968

Signature of Author [illegible]
Department of Electrical Engineering, May 17, 1968

Certified by James D. Bruce
Thesis Supervisor

Accepted by
Chairman, Departmental Committee on Graduate Students

A SPECIAL-PURPOSE LIST PROCESSOR

by

BURTON JORDAN SMITH

Submitted to the Department of Electrical Engineering on May 17, 1968, in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

The relationships between data bases, data retrieval, and sequential representations of data bases is explored. A programming language to facilitate the manipulation of lightly structured sequential representations of a data base is described. The language is imbedded and does not reuse storage. A language manual and a sample program are included.

Thesis Supervisor: James D. Bruce

Title: Associate Professor of Electrical Engineering

ACKNOWLEDGMENT

To Professor James D. Bruce, for his interest, insight, and guidance;

To Arthur Bushkin, for many fruitful and interesting discussions;

To Dottie, for her patience as a typist and her sympathy as a wife.

B. J. S.
Belmont, Massachusetts
May, 1968

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1
INTRODUCTION

The Electrical Engineering Department at M.I.T. has been using the Compatible Time Sharing System (CTSS)[1] for some time in conjunction with the administrative activities of the department. A programming language has been developed within CTSS in an attempt to help solve many of the data manipulation problems encountered in the daily course of departmental operations. It is expected that the language will grow in power and flexibility as more experience is gained with it; so far, it has proven to be a useful administrative tool.

The first employment of CTSS in Electrical Engineering department administration arose in connection with a need for better information about the activities of a growing faculty. An information and control system was devised to meet this need. In this sytem, a CTSS context editing program, TYPSET[1], is used to create and maintain a textual format disc file containing the departmental teaching assignments for the coming term. The file is printed using a memorandum generator, RUNOFF[1], which is compatible with TYPSET; the result is a printed listing indistinguishable from manually typed matter. This listing is then duplicated and mailed to the teaching staff of the department for comment. Using TYPSET, changes resulting from these comments are inserted into the file in preparation for the next mailing. Three iterations are performed, and by the time the term starts, the file reflects quite reliably the true activities of department personnel.

The existence of an accurate data base encouraged efforts to obtain other kinds of information from it. A program was written by an M.I.T. graduate student to generate a list of subject offerings together with a breakdown of the activities of the faculty and staff engaged in teaching each subject. The list is compatible with TYPSET and RUNOFF, and is appended to and mailed with the teaching assignments from which it is generated.

A third list is maintained by the department. This is a department committee membership list and is updated through TYPSET, although it could and should certainly be generated from the data base. In fact, the programming language described below is designed to facilitate the creation of computer programs to accomplish such tasks.

In the next chapter, some theoretical considerations relevant to data management in general are discussed. Chapter 3 gives a motivation for the language in the light of the theory and an overview of the system architecture. Chapter 4 deals with the commands and conventions of the language, and suggestions for applications and extensions together with concluding comments are to be found in Chapter 5. A user's manual is included as Appendix A, and a programming example appears as Appendix B.

CHAPTER 2

THEORETICAL CONSIDERATIONS

A. Definition of a Data Base.

In order to facilitate discussion, it will be helpful to explain what is meant by the term <u>data base</u>. A data base will be defined as an arbitrary set of symbols D called the <u>data</u>, a function T from an index set I of <u>data types</u> to the power set of D, and a function B from an index set N of <u>relation names</u> to a set of relations defined in cartesian products over certain of the indexed subsets of data. Formally, a data base is a 5-tuple (D, I, T, N, B), where

$$T : I \rightarrow 2^D \qquad (1)$$

$$B : N \rightarrow \{R_J \subseteq \bigtimes_{i \in J} T_i \text{ for all } J \subseteq I\} \qquad (2)$$

(The symbols used in this chapter for set-theoretic concepts conform with common usage[2,3] insofar as possible. Equations referred to in the text will be enclosed in parentheses.)

The data types need not uniquely name subsets of data, since it is desirable to allow a subset of D to recur in a cartesian product, as in the case of a binary relation in $T_i \times T_i$ for $T_i \subseteq D$. For this reason, $T_i = T_j$ does not imply that $i=j$, and hence i and j both name T_i. Another way of saying this is to say that the function T is not one-to-one. The function B is one to one, however, and hence elements of N uniquely name certain of the relations in the range of B. The elements of I may be grouped in equivalence classes by T^{-1} to uniquely name the subsets of D, but this will not be necessary in the development. Some examples of data types might be man, house, and year; examples of relation names might be lives in and has owned since.

It is important to distinguish between what is in the data base and what is not. The elements of D are symbols, i.e. names, representing entities or objects external to the data base. The named entities are interrelated via external n-ary relations which are represented in the data base by B_n and collectively by B. If $i \in I$, then T_i names a subset of the set of entities; if $n \in N$, then both n and B_n name an external n-ary relation among entities. Thus, a data base is a model of reality rather than reality itself, and it is therefore appropriate to examine the applicability of the model and to thereby explore its properties.

Consider a collection of real entities which are linked via several n-ary relations. In general, the relations among the entities will recur; for example, the entities Steve and Sue may be related in the same way as the entities John and Joan. In such a case, it is natural to form subcollections which have as members all of the entities which relate to other entities in the same way. The family of all such subcollections is represented by the function T in the data base; the entities in each subcollection we identify by an element of I, and T maps the element to the subset of D which represents the subcollection. In Figure 1, the symbols John and Steve are of type man, the symbols Joan and Sue are of type woman, and certain members of the image T_{man}

are related to certain members of the image $T_{\underline{woman}}$ by the binary relation is married to which is represented by the edges between John and Joan, Steve and Sue.

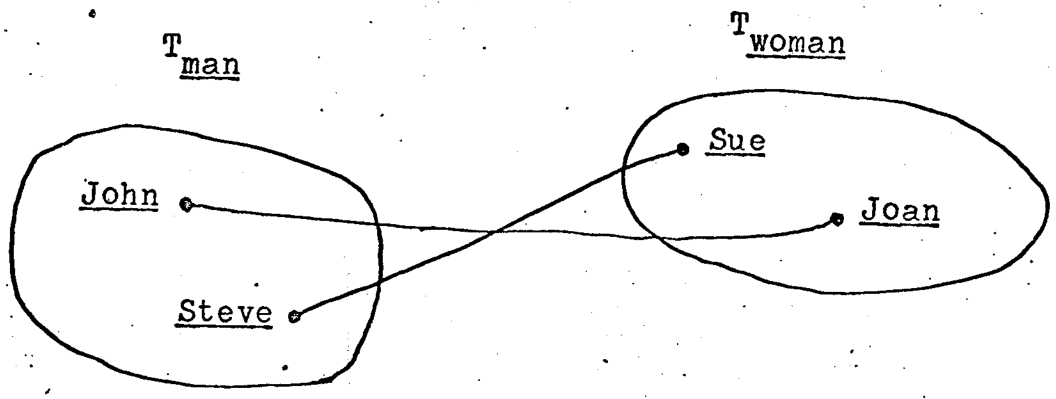


Figure 1. Two related subsets of data.

It will be convenient to represent the existence of relations between subsets of data graphically. The fact that a relation exists between two subsets is all that we are interested in, and hence we represent the relations B_n, $n \in N$ by edges and the subsets of data T_i, $i \in I$ by vertices. If B_n is ternary, or worse, the edges become "hyperedges" which connect three or more vertices. In Figure 2, a binary or normal edge connects $T_{\underline{man}}$ and $T_{\underline{house}}$, whereas a ternary hyperedge connects $T_{\underline{man}}$, $T_{\underline{house}}$, and $T_{\underline{year}}$. The binary edge represents lives in and the ternary hyperedge represents has owned since. We will call graphs with hyperedges "hypergraphs" to distinguish them from conventional graphs,[4] and refer to the "hypergraph associated with a data base."

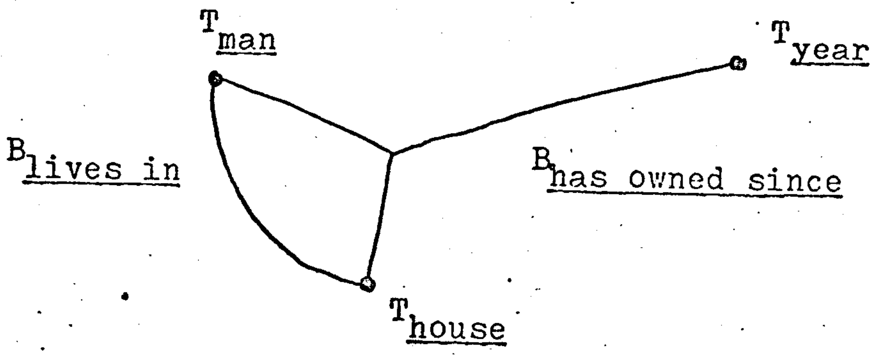


Figure 2. Example of a hypergraph.

Intuitively, a data base represents something more specific than just a collection of entities and a collection of relations. If there exists a totally incestuous collection of entities and relations such that every subcollection taking part in a relation in the same way is totally included in the collection, then the collection is somehow self-contained and should properly be represented by its own data base. Two such totally incestuous collections should not be represented in the same data base, since each is a self-contained object and no additional meaning would result from the conjunction. Figure 3 depicts two such totally incestuous collections; the hypergraph is in some sense not connected, because no relation exists between any of $T_{\underline{man}}$, $T_{\underline{house}}$, $T_{\underline{year}}$ and either of $T_{\underline{element}}$, $T_{\underline{valence}}$. The hypergraph of Figure 3, therefore, represents not one data base but two.

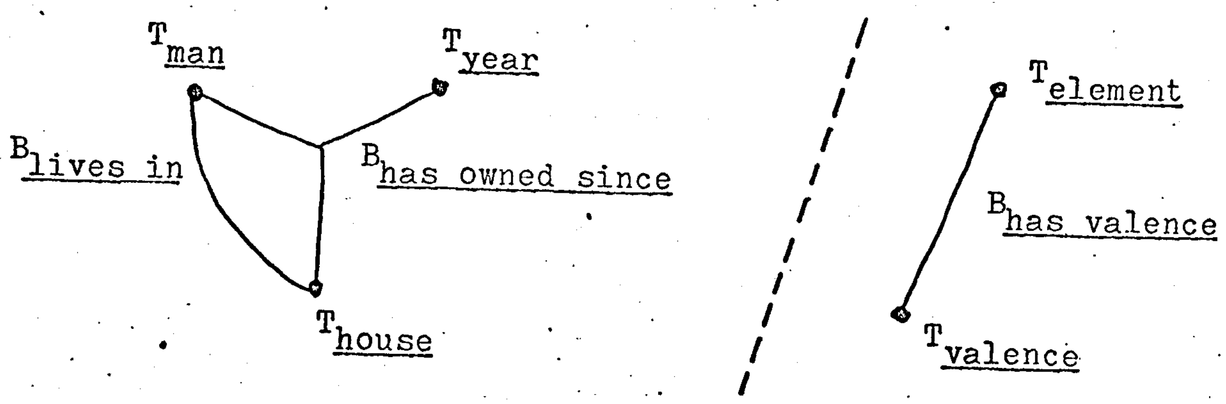

$T_{\underline{man}}$
$T_{\underline{year}}$
$B_{\underline{lives\ in}}$
$B_{\underline{has\ owned\ since}}$
$T_{\underline{house}}$
$T_{\underline{element}}$
$B_{\underline{has\ valence}}$
$T_{\underline{valence}}$

Figure 3. Non-connected hypergraph.

In order to formalize the connectedness property, it will be useful to introduce the notion of a projection. Let an n-ary relation be defined in the cartesian product over an indexed family $\{T_j\}$, $j \in J$. If R_J is the relation under discussion, then

$$R_J \subseteq \bigtimes_{j \in J} T_j \qquad (3)$$

For every subset of J, a projection of R_J onto the subset is defined. Denote the projection of R_J onto $K \subseteq J$ by R_K. R_K is a relation; in particular,

$$R_K = \{X : (\forall i)(\exists t_i)(t_i \in T_i \wedge \bigtimes_{i \in J} \{t_i\} \subseteq R_J$$

$$\wedge\ X \in \bigtimes_{i \in K \subseteq J} \{t_i\})\} \qquad (4)$$

Certain properties of the projection operation are obvious. Since R_K is a relation, it may be further projected on subsets of K, and projection is transitive; if $L \subseteq K \subseteq J$, then the projection on L of the projection on K of R_J is equal to the projection on

L of R_J. If L is empty, then the projection of R_J on K is the empty relation; if K = J, the projection of R_J on K is equal to R_J. For an example of projection, let J = {man, house, year}, K = {man, house}, T_{man} = {John, Steve}, T_{house} = {2 Shady Lane, 115 Main Street, 397 Center Avenue}, T_{year} = {1955, 1961}. Then if R_J = {(John, 2 Shady Lane, 1961), (John, 397 Center Avenue, 1955), (Steve, 115 Main Street, 1961)}, R_K = {(John, 2 Shady Lane), (John, 397 Center Avenue), (Steve, 115 Main Street)}.

Another property of projection is that an n-ary relation cannot always be reconstructed from the set of projections on proper subsets of its index set. For example, the ternary relation {(A, I, P), (A, 2, Q), (B, I, Q), (B, 2, P)} has the same unary and binary projections as the cartesian product {A, B} $\times$ {1, 2} $\times$ {P, Q}.

The definition of connectedness for a data base can now be made precise. A data base is connected if and only if the hypergraph associated with the data base is connected, which is true if and only if the graph of binary projections is connected. The graph of binary projections is the graph associated with the data base after all n-ary relations have been replaced by their binary projections. Figure 4a depicts the hypergraph of previous examples; Figure 4b depicts the graph of binary projections obtained from the same data base.

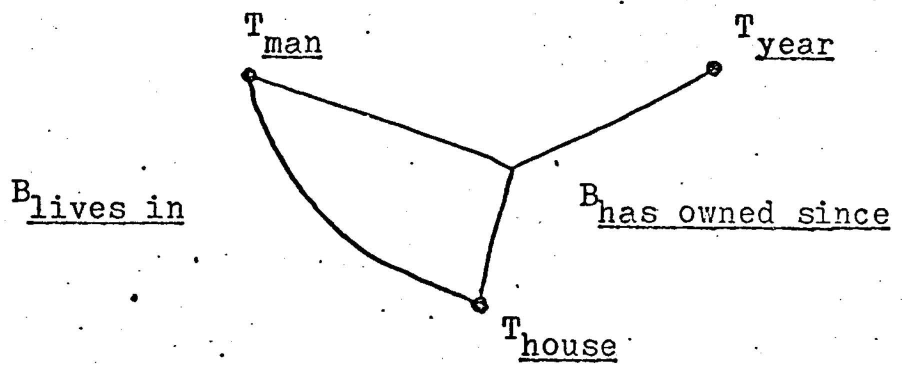


Figure 4a. A hypergraph.

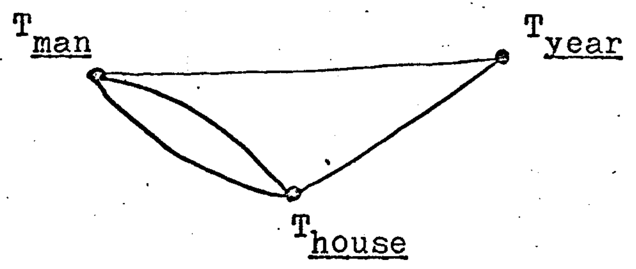


Figure 4b. Its graph of binary projections.

The concept of a data base has been defined, and certain of its properties discussed. It will be our next task to discuss an operation on the model which intuitively correponds to one which has classically been performed on data bases, namely the retrieval operation.

B. Retrieval on a Data Base.

Implicit in the discussion so far has been the idea that subsets of data T_i may and often must take part in more than one relation to satisfy the connectedness property. By "taking part" we mean that a subset of data is a unary projection (technically, a union over the projection) of the relation it "takes part" in. The connectedness property was based on intuitive notions of what a data base should be. One result of this property is that there exists at least one sequence of binary projections of relations which, when represented on the graph of binary projections, form a chain between any two subsets of data. Since the graph of binary projections may not be a tree in general, more than one such chain may exist, and a way of naming chains is required. Each relation is uniquely named by the elements of n, but binary projections of the relations may well not be, since an n-ary relation has $\binom{n}{2}$ binary projections. It will be convenient to use subsets of I to specify the binary projection in question, and indeed, the set of indices onto which a relation is projected together with the name of the relation fully characterize the projection. If B_n, $n \in N$ is a relation in $\bigtimes_{i \in J} T_j$ for $J \subseteq I$, then the projection of B_n on $K \subseteq J$ is specified by n and K without ambiguity and may be represented by the ordered pair (n, K). Furthermore, since the relation B_n is merely (n, J), we need not consider relations and can deal only with their projections. This shall be our approach. Figure 5 shows a graph of binary projections named according to this scheme.

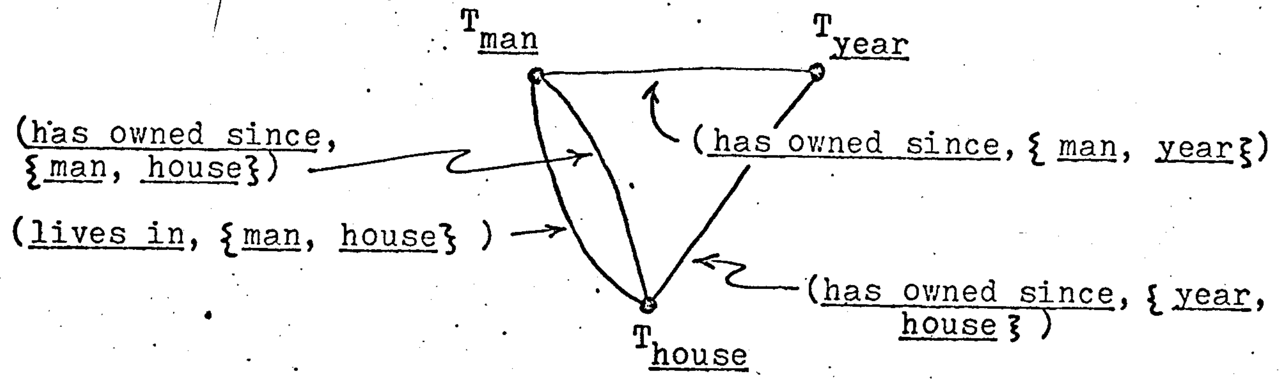

$T_{\underline{man}}$
$T_{\underline{year}}$
$T_{\underline{house}}$
(has owned since, {man, house})
(lives in, {man, house})
(has owned since, {man, year})
(has owned since, {year, house})

Figure 5. Graph of named binary projections.

A chain in the graph of binary projections may now be specified by a sequence of projection names. The chain may be composite; that is, it may traverse an edge more than once. The chain may be infinite and be specified by an infinite sequence of projections; if the graph is finite, all infinite chains are composite. If a chain is not composite, it is simple. The simple chain specified by the projection sequence (<u>lives in</u> {<u>man</u>, <u>house</u>}), (<u>has owned since</u>, {<u>year</u>, <u>house</u>}) is shown in Figure 6.

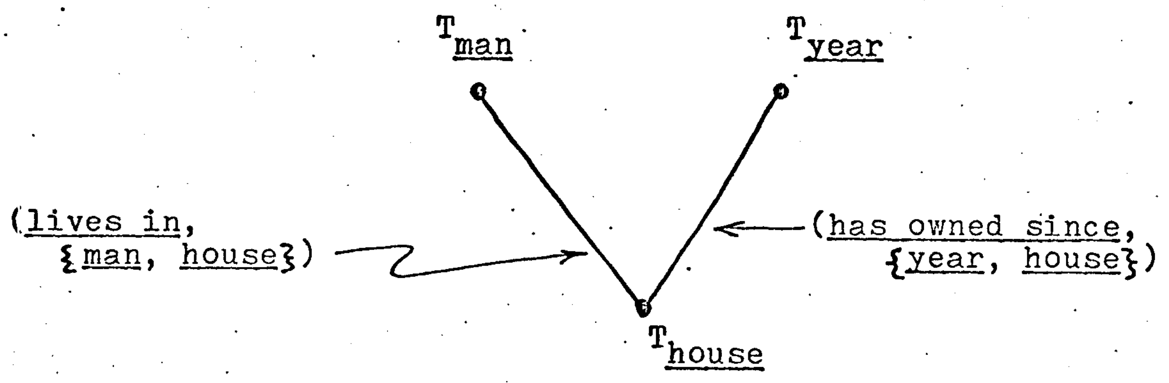


Figure 6. Simple chain in the graph of Figure 5.

The binary projections defined by a chain may be combined by composition into a single relation. This possibility motivates the definition of binary retrieval. A binary retrieval request consists of a singleton $(i, \{t_i\})$, $t_i \in T_i$, and a chain of binary projections which when composed yield a binary relation in $T_i \times T_j$. The result of the retrieval is the image of $\{t_i\}$ under the composed relation and is a subset of T_j. For example, the binary retrieval request consisting of (man, {John}) and (lives in, {man, house}), (has owned since, {year, house}) will result in an image set containing the dates since which the houses John lives in have been owned. (Do not draw the conclusion that John owns the houses he lives in.) More complex procedures are possible, e.g. logical operations on binary retrievals with the same chain terminus might result in the set-theoretic analogues on the images, with negation applied to an image $A \subseteq T_j$ having the obvious meaning $T_j - A$. Since composite and infinite chains are allowed, binary retrieval is quite versatile. It is the case, however, that binary retrieval is not adequate to deal with n-ary relations, and is therefore not capable of exploring all of the relations among data, simply because some of the structure of the data base may be lost when it is replaced by its binary projections. A more powerful concept is needed.

A retrieval in the most general sense should involve the specification of some data and some external relations among the data together with a rule for transforming these relations into another relation which represents the result of the retrieval.

The kinds of rules which are allowed specify completely what a retrieval is. We shall assert that a legal retrieval rule may only invoke the operations of composition, projection, union, intersection, and complementation. The operands may be external relations, relations internal to the data base, or data.

A binary retrieval is clearly a retrieval, as are all logical combinations of binary retrievals. Consider now a data base consisting of a single ternary relation among names, addresses, and telephone numbers, and we desire the telephone numbers for a name-address pair. We would like to form the "composition" of the input relation (the name-address pair) with the ternary relation of the data base to obtain the (unary) relation of the telephone numbers. We need a more generalized notion of composition to be able to do this, one that is defined for n-ary relations. Note that the results are not necessarily the same if simple retrievals are used, as has been pointed out. The generalized composition operation should include binary composition as a special case, and should be such that the application of an n-ary relation to an m-ary relation should be an (n-m)-ary relation for m less than or equal to n and undefined otherwise. Further, the tuples of the (n-m)-ary relation should be such that when they are "combined with" some element of the m-ary relation, they yield an element of the n-ary relation. We denote composition of two relations by juxtaposition as is customary for binary relations.

Let R_J and S_K be two relations

$$R_J \subseteq \bigtimes_{j \in J} T_j \qquad (5)$$

$$S_K \subseteq \bigtimes_{k\in K} T_K \tag{6}$$

and let some subset of I be nonempty and

$$\bigcup_{\ell\in L}\{T_\ell\} = \bigcup_{j\in J}\{T_j\} - \bigcup_{k\in K}\{T_k\} \tag{7}$$

Then the composition $R_J S_K$ is defined, and

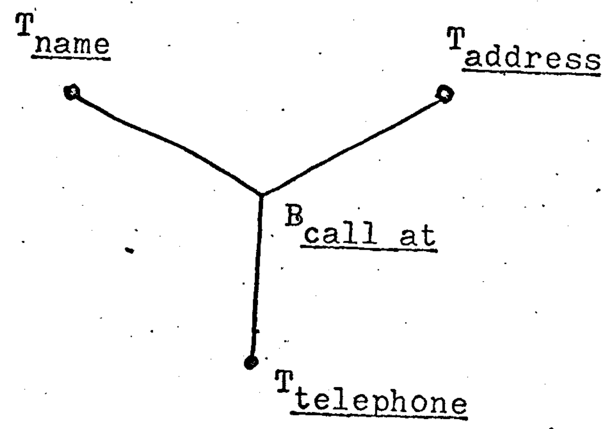
$$R_J S_K = \{X:(\forall i)(\exists t_i)(t_i \in T_i \wedge \bigtimes_{i\in J}\{t_i\} \subseteq R_J$$
$$\wedge \bigtimes_{i\in K}\{t_i\} \subseteq S_K \wedge X \in \bigtimes_{i\in L}\{t_i\})\} \tag{8}$$

It should immediately be clear from the similarity of the definitions that projection and composition are closely related. In fact, if $L \subseteq J$, then the projection of R_J on L is merely

$$R_L = R_J(\bigtimes_{i\in J-L} T_i) \tag{9}$$

The composition operator is neither commutative nor associative in general; if $J-K \subset L \subset J$ and $K \subset J$, $R_J S_K$ and $Q_L(R_J S_K)$ are defined whereas $S_K R_J$ and $(Q_L R_J)S_K$ are not. Other pertinent facts are that if $\{\phi\}$ is the empty relation, then $R_J\{\phi\} = R_J$ and $\{\phi\}R_J$ is undefined for J nonempty, and that $R_J R_J = \{\phi\}$. These properties are immediate results of the definition.

Consider the previous example involving names, addresses, and telephone numbers. The data base contains one ternary relation, and its hypergraph is shown in Figure 7 together with a tabulation of the relation, which we shall name call at.



$^{T}\underline{name}$
$^{T}\underline{address}$
$^{B}\underline{call\ at}$
$^{T}\underline{telephone}$

$^{B}\underline{call\ at}$:

(John, 397 Center Avenue, 515-2347)

(John, 397 Center Avenue, 515-2348)

(John, 2 Shady Lane, 506-1299)

(Steve, 115 Main Street, 515-2348)

(Steve, 397 Center Avenue, 506-1299)

Figure 7. An example for retrieval.

We wish to explore several kinds of retrieval on this data base using the five operations and, or, not, projection, and composition. Since $^{B}\underline{call\ at}$ is merely (call at, { name, address, telephone}) as discussed on page 16, we so replace it, abbreviating the indices to (c ,{n,a,t}). External relations which serve as input will be similarly abbreviated.

The first retrieval to be considered is binary, and might be described as "get all telephone numbers where name is John and address is 397 Center Avenue." We construe this to mean $(c,\{n,t\})(x,\{n\}) \cap (c,\{a, t\})(y,\{a\})$. That is, two binary projections of the call at relation are to be composed with external unary relations x and y, and the results intersected. We have that (c,{n,t}) is {(John, 515-2347), (John, 515-2348), (John, 506-1299), (Steve, 515-1248), (Steve, 506-1299)} and hence the set to the left of the intersection is {(515-2347), (515-2348), (506-1299)}. (The indices should have rightfully been included in the projections but have been left out for brevity.) Proceeding similarly, the right side of the intersection is {(515-2347), (515-2348), (506-1299)}, so that the result of the retrieval is {(515-2347), (515-2348), (506-1299)}.

Now let us consider the retrieval verbalized as "get all telephone numbers where (name, address) is (John, 397 Center Avenue)." We interpret this as (c,{n,a,t})(x,{n,a}) and compose the call at relation itself with the external binary relation x. The result is the unary relation {(515-2347), (515-2348)} and is not the same as the result obtained from the binary retrieval. If we attempt to construct a binary retrieval which will give the same results, such as "get all telephone numbers where name is John and not name is not John and address is 397 Center Avenue and not address is not 397 Center Avenue," we will obtain yet another result in general; in this case, it happens to be {(515-2347)}, the telephone number that John and 397 Center Avenue share with each other and with nothing else, which is of course different from the other two.

The example was designed to show that retrieval in the general sense is necessary to perform certain well-defined and interesting operations on a data base, and that binary retrieval is not always adequate to perform these operations. Rather than discuss more completely the notion of generalized retrieval, the next section will deal with a somewhat different topic: The problems involved in representing a data base in sequential form.

C. Data Streams.

Bushkin[5] has described in detail how a data base represented by an acyclic directed graph can be converted reversibly into a sequence of data called a data stream. The acyclic directed graphs which he describes correspond in our model to hypergraphs containing exactly one hyperedge together with a well-ordering on the set of vertices $\{T_i\}$ of the hypergraph. Each data subset T_i is also the domain of a well-ordering, so that the elements of the data stream are triples consisting of a datum, the ordinal number of the datum under the well-ordering on the data subset to which it belongs, and the ordinal number of the subset under the well-ordering on all subsets. Figure 8 represents such a hypergraph with ordinally numbered vertices; Figure 9 depicts the ordinally numbered data in each subset (vertex) from Figure 8, together with the relations among them.

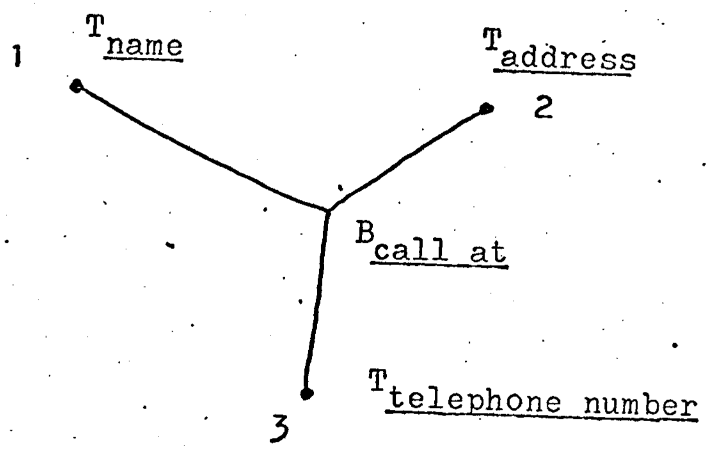


Figure 8. Hypergraph with ordinally numbered vertices.

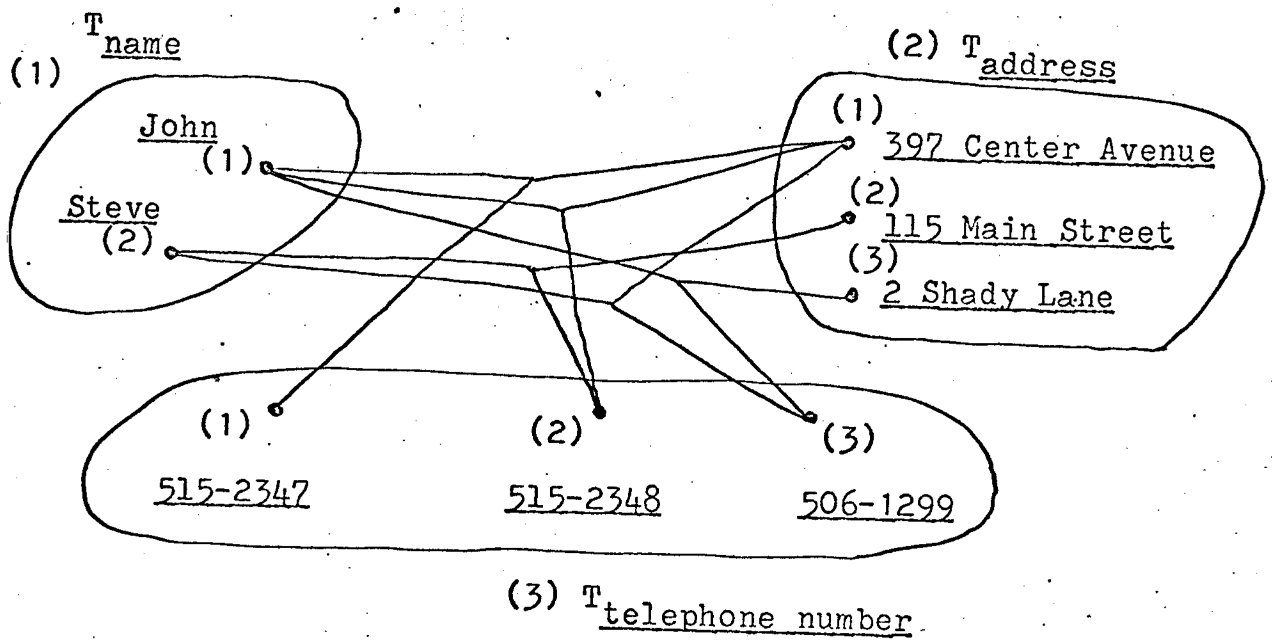


Figure 9. Ordinally numbered related subsets.

The data base is converted into a data stream by the lexicographic order generated by the ordinal numbers on data and subsets of data applied to the n-tuples of the relation. For the example of Figure 9, the data stream which results is shown in

Figure 10. The typographic form of the figure is not important; the structure is totally conveyed by the datum—number triples and by the order of the sequence.

John	1,1
397 Center Avenue	2,1
515-2347	3,1
515-2348	3,2
2 Shady Lane	2,3
506-1299	3,3
Steve	1,2
397 Center Avenue	2,1
506-1299	3,3
115 Main Street	2,2
515-2348	3,2

Figure 10. Data stream resulting from Figure 9.

If all data bases could in fact be represented by single n-ary relations, then the problem of stream generation from a data base would always be soluble by Bushkin's method. It is in fact possible to represent data bases of more complex structure by single n-ary relations if a certain amount of artificiality can be tolerated. There are excellent pragmatic reasons for wanting to be able to represent a data base by a data stream, so that artificiality may not be too high a price to pay. For example, in representing a data base in a digital computer, a well-ordering is provided on the elements of the representation, since almost all storage devices are sequentially addressable. If the data base is represented in stream form, the representation can be made very compact; it is therefore important to attempt to represent data bases by single n-ary relations.

Consider an n-ary relation which has been projected onto several of its index subsets in such a way that the associated hypergraph is still connected. Then the result represents a data base of some sort with rather special properties. The process is not reversible in general; many data bases associated with n-vertex hypergraphs cannot be so constructed. For example, consider a 2-vertex hypergraph containing two binary edges. If the relations which the edges represent are different, then clearly no binary relation can be projected to yield the two distinct binary relations. It can be shown that it in fact is possible to so represent a data base when the data base is associated with a hypertree, and that any data base associated with a hypertree of n vertices can be constructed by projection from a single n-ary relation of some sort. A hypertree will be defined as a hypergraph which is connected but loses this property if any hyperedge is deleted. Other definitions are possible, but would require additional properties of hypergraphs which have not been discussed.

We will sketch an indirect proof of the assertion that a data base representable by an n-vertex hypertree can be created by projection from a single n-ary relation over the same subsets of data. Suppose that some relation $B_n \subseteq \bigtimes_{i \in J} T_i$ in a data base characterized by a hypertree cannot be represented by a projection R_J of some n-ary relation as described above. That is, there is at least one element that R_J and B_n do not have in common, an element that R_J must or must not have regardless of the constraints imposed by B_n. This implies that some other relation in the data base further restricts R_K and in particular, for some subset K of J, there is a relation $B_m \subseteq \bigtimes_{i \in K \subseteq J} T_i$ also in the data base. Now remove B_m from the data base, and consider the graph of binary projections. If this graph was connected before deletion of B_m, it still is, since $K \subseteq J$ implies that any pair of vertices connected by a binary projection of B_m are still connected by a binary projection of B_n. Since the graph of binary projections is still connected, the hypergraph associated with the data base is still connected and could not have been a hypertree.

The above result justifies the creation of data streams from any data base characterized by a hypertree. We can extend this facility further by introducing yet another artifice. If duplication of subsets of data is allowed, then any data base whatsoever can be converted into a data base characterizable by a hypertree merely by splitting vertices (duplicating subsets) until the hypergraph satisfies the hypertree property.

An example of the process of converting a data base to a data stream is depicted in Figure 11a-c. In Figure 11a, the hypergraph of a data base is shown. Sufficient vertices have have been duplicated in Figure 11b to convert the hypergraph to a hypertree; in Figure 11c, the hypertree has been converted to a single n-ary relation and the vertices ordinally numbered in preparation for the stream generation procedure.

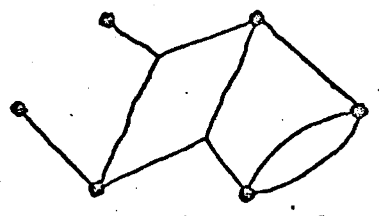


Figure 11a. Hypergraph of a data base.

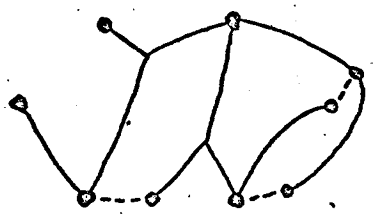


Figure 11b. Hypertree from Figure 11a.

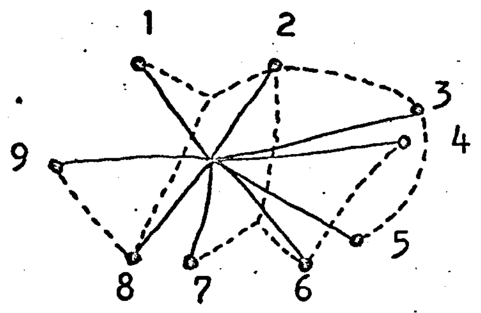


Figure 11c. Single relation from Figure 11b.

We have not described how a particular data base is to be converted into a data stream. The method by which vertices to be duplicated are chosen, procedures for changing hypertrees to single relations, and rules for ordinally numbering data and subsets of data have not been our concern. In the next chapter, these problems will be discussed in the light of a particular implementation: a language to manipulate data streams.

CHAPTER 3

OVERVIEW OF THE LANGUAGE

A. Stream Manipulation.

In the context of digital computation, data bases are often represented by data streams. One of the reasons for this has already been mentioned; data streams can be stored efficiently in sequential devices. Communication channels are often sequential as well, so that communication of a data base is efficient when the data base is in data stream form. Unfortunately, a great deal of structure must be artificially imposed on a data base to obtain a data stream, with the result that the structure often gets in the way and must be modified or removed. In particular, it is often convenient to respecify the order on the collection of subsets $\{T_i\}$ to generate "inverted" streams; one way of doing this is to recreate the single n-ary relation from which the stream was generated. Another operation involves deletion of certain of the data subsets T_i from the data stream entirely. For this operation, it is not necessary to recreate the n-ary relation; it is adequate merely to detect the type and delete those not desired. The programming language to be described in the balance of this thesis was designed to facilitate both of these operations.

One primary facility of the language is the ability to recognize and order the data contained in the stream. The order relations among and within data subsets are partially destroyed in the stream generation process and must be reconstructed to enable generation of an inverted stream. How the order relations are actually computed will be discussed later; suffice it to say that the order on each subset T_i is represented by a one-way list which threads through all of the data in the subset. Such a structure is shown in Figure 12. Each horizontal rectangle symbolizes a block of one or more consecutive words in the high-speed core memory of CTSS, and represents one element from a data stream.

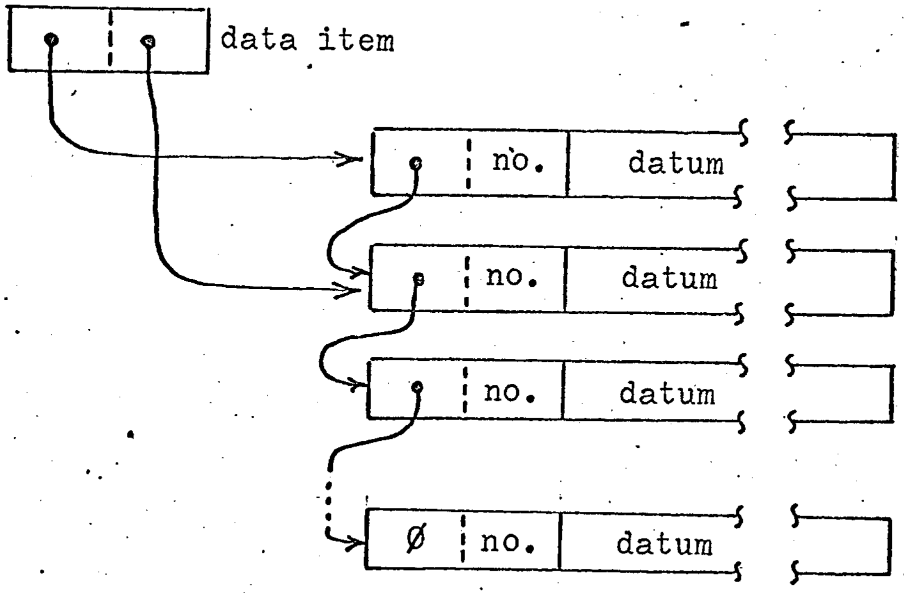


Figure 12. List of data.

The leftmost word in the representation of each stream element contains a pointer to the next element and a number signifying how many machine words are used to represent the datum. If that number is n, then the next n words contain a bit pattern which represents the datum in some way. The order on the data is given by the list, in that any datum reachable from another datum is in the image of that datum under the order relation. The end of the list is signified by a zero.

The box in the upper left-hand corner of Figure 12 labeled "data item" represents a single machine word containing two pointers. One pointer points to the top of the list; the other points to some element in the list. That is, a data item points to a stream element and to the list of similar stream elements, and thereby indicates the datum, the subset to which it belongs, and the order in that subset. A data item is generated whenever a stream element is found and inserted in a list, and is used to represent the datum and the current state of its list. As the stream is read, data are extracted from the stream and routed to the appropriate points in the appropriate lists. The data items are returned in a sequence matching the order of the data stream.

If stream inversion is to be performed, then the sequence of data items which was created from the input data stream must be converted into a set of objects representing an n-ary relation from which an inverted output stream may be generated. This is accomplished by selecting certain of the data items from the sequence and generating a list of n-tuples of these items. Such a list of n-tuples is shown in Figure 13. The n data items comprising each new tuple are stored sequentially and appended to the bottom of the list of n-tuples. In most respects the list is similar to the lists of data shown in Figure 12, except that data items rather than data are stored.

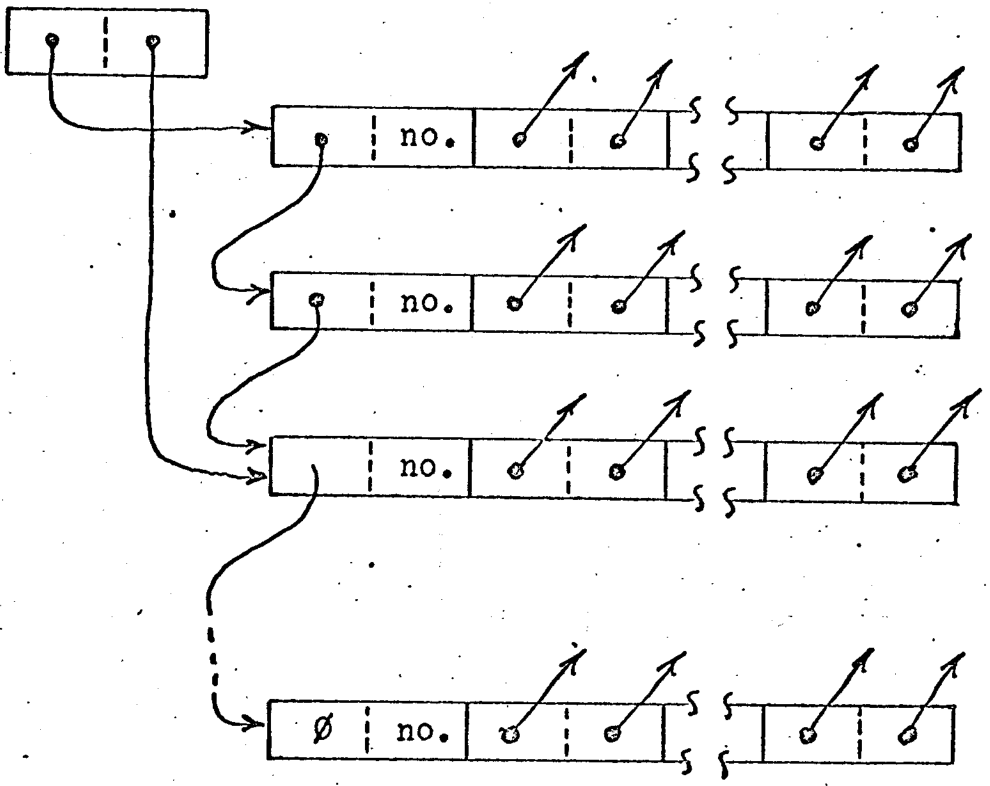


Figure 13. A list of n-tuples of data items.

The data items in a given position in each n-tuple must point to data in the same list, in order that the lexicographic order used to generate the output data stream will be well-defined. The lexicographic order is generated by a multiple-pass sorting operation on the list of n-tuples. The list of n-tuples is first sorted according to the order of the list of data pointed to by the rightmost component of each n-tuple. When this has been done, the next component to the left is taken and the list of n-tuples resorted according to the list of data pointed to by that component.

The sorting terminates when the leftmost components of the n-tuples have been sorted on, and the output stream can now be generated. Note that the new order relation among data subsets is conveyed by the left-to-right order of data items within each n-tuple, whereas the order relation within each subset is conveyed by the lists of data pointed to by those data items.

Creation of the output stream from the sorted list of n-tuples is accomplished by outputting each n-tuple from the list in the new order specified by the list, splitting each n-tuple into its constituent data items. Each output of an n-tuple results in a transfer of program control to one of n locations in the program, depending on the most significant component of the n-tuple which changed since the last n-tuple output; this facilitates certain kinds of computations which may be desired just before and during the output process. For example, the stream can be outputted in an "outline form" similar to that shown in Figure 10 through the use of this facility. Appendix B shows an example of such an output specified in the language.

B. String Operations.

It is often the case that data streams are not actually represented by triples of data and ordinal numbers but by strings or sequences of strings. Strings are commonly used to represent data, and in fact it may be difficult to conceptually separate the two ideas in some situations. Strings are in fact more complex than the data which they represent, in that a datum is a symbol whereas a string is a sequence of symbols. The datum John might well be represented by the string 'John,' but 'John' is a sequence of symbols (J,o,h,n) and intuitively contains somewhat more information. It is meaningful to say that 'John' contains a 'J', but in no sense does the datum John contain anything. We shall represent data by strings, and operate under the assumption that all data are represented by strings both internally and externally. Thus, the bit patterns representing data in each data item are strings (i.e., bit patterns representing strings) which are interpretable by the computer.

The ordinal numbers contained in each stream element could also be represented by strings, or perhaps by integers (bit patterns representing integers). The most interesting case, however, occurs when one or both ordinal numbers are represented by some computable property on the strings which represent the data. The familiar alphabetic ordering process applied to a set of strings results in a natural and intuitively appealing order on the set; this order is in fact a well-ordering and can be computed from the set of strings and an order relation on the characters. Similarly, the subset T_i to which a datum represented by a string belongs can often be computed from the context of the string; if a name string such as 'John' always appears in a particular way relative to other strings, then that property can be used to characterize the set of all name strings. This is nothing more nor less than a parsing operation; if the input strings were English text, the problem might be to recognize all of the verbs, nouns, prepositions, etc. Whether a string is classifiable by context or by some other procedure is pertinent; we shall stipulate that the input strings representing a data stream must be reducible by a context-sensitive parser to data items. This is in fact not a very restrictive condition, considering the kinds of inputs that are of interest.

As an example, let a telephone directory serve as input. Assume that every name contains only letters, every address begins with numbers and contains both numbers and letters, and every telephone number contains only numbers. The lines of the telephone directory can be parsed into names, addresses, and telephone numbers unambiguously by requiring that a telephone number have no right context and contain seven numerals, and then that a name have no left context and have one or more numbers as right context. The names, addresses, and telephone numbers can then be ordered alphabetically, by table look-up, or by some combination of these two operations. Thus, a telephone directory in natural form can be interpreted as a data stream by the language, and inversion and subsetting of that data stream becomes possible.

The strings which represent data in the input must be placed in the appropriate lists of data items at the appropriate points, in order that the data items resulting from the parse will be defined. One technique is to let every list of data be wholly maintained by the parser, and this is in fact done. The parser consists of a set of independent subroutines, each of which maintains one or more lists of data. The subroutines fetch input strings from a standard location and update the appropriate lists, returning data items if the input string was successfully parsed.

In the next section, some considerations relating to the design of the language are discussed. The description of the language up to now should make clear the fact that it is designed to interpret and manipulate loosely specified, textually formatted lists. The name of the language is SPLP, for Special Purpose List Processor, primarily because the language is highly optimized for this task and is unsuitable for general-purpose computing, and because it operates on lists. The language is not in fact a list processing language as the term is commonly meant, but several issues that arose in the design of SPLP are pertinent to list processing languages.

C. Design Considerations.

The SPLP language is imbedded in MAD[6] and consists entirely of subroutines. In this respect, it is similar to SLIP[7]. The reasons for imbedding were several. First, an imbedded language is easy to write; most of the onerous details are handled by the parent compiler. Second, the language is easy to modify; the subroutines may be individually compiled, and new features may be added to the language easily, either by introducing new subroutines or modifying old ones. Third, the combination of the parent language and the imbedded language is usually more powerful than either language alone.

In selecting a language in which to imbed, the most valuable characteristic sought is simplicity. In particular, languages which do extensive automatic type conversion or maintain large run-time stacks in available storage are often unsuitable. There is no convenient facility whereby an imbedded language can determine what the parent language has done to arguments or to available storage; the best solution is to imbed in a language which modifies neither. MAD and FORTRAN are very good in this respect.

All of the lists generated by SPLP from the input stream are stored in high-speed core. Since the lengths of the items are variable, reuse of storage poses a difficult (and classical) problem. Happily, the special nature of the applications for which SPLP was designed enabled a simple solution. The stream inversion operation involves reading the entire input stream, sorting the lists obtained, and then generating the output stream. Virtually the only demands for additional storage space are made during the input process, but no recovery can be made until the output process begins. For this reason, it was decided that no garbage collection scheme would be incorporated into SPLP, since very little or no reuse of storage would be possible.

One of the more important requirements imposed on SPLP was that it be very efficient. The language is designed to facilitate a class of operations which are commonly performed by administrators and others in the daily course of affairs, and the economics of the situation are such that a small increase in program efficiency will result in a fairly large saving in administrative cost. The efficiency requirement motivated several decisions relating to the design of the language.

First, very little checking for programming errors is done by the language. In particular, calling sequences are usually not checked either for content or for length. It was felt that the additional overhead required for such checking was excessive. Since in most SPLP subroutine calls the fact that an argument is zero has a well-defined and natural interpretation, a highly likely source of possible programmer error is avoided: the problem of uninitialized arguments.

Second, most subroutines taking variable length calling sequences obtain the length of the sequence from the first argument. That is, if n variables are to be passed to such a subroutine, then the first argument in the calling sequence is n and the variables make up the next n arguments, making the calling sequence of length n + 1. Once again, it was felt that one calling sequence length processor call for every subroutine call was excessive.

Third, the SPLP subroutines are written in assembly language. This fact enables efficient use to be made of index registers in the operations involving lists, and allows maximum utilization of machine resources in general.

In summing up, SPLP consists of a set of assembly language subroutines callable from MAD programs. Operations on input strings result in lists of strings which can be sorted and outputted in a prescribed sequence. In the next chapter, the subroutines which make up SPLP will be discussed in detail. A further discussion of each subroutine from a programmer's point of view may be found in Appendix A. A familiarity with CTSS is assumed.

CHAPTER 4
DESCRIPTION OF THE LANGUAGE

A. Supervisory Routines.

Three SPLP subroutines perform the necessary supervisory functions for the language. The first of these, BEGIN, performs certain preliminary tasks and must be the first SPLP subroutine called in a program. BEGIN first leaves multiple tag mode and sets one level of interrupt, making the interrupt return the entry point of the closing routine ENDOUT. This allows program escape to CHNCOM, rather than DORMNT, to be initiated from the console. BEGIN then fetches and saves the current command buffer, and searches the command buffer for the presence of the argument (DBUG). If (DBUG) is present, the loader is called to load FAPDBG. Memory bound is then extended to the top of core, and the complement of the next available storage location (i.e., the old memory bound) is placed in index register 7. Index register 7 must always contain the complement of the next free storage loaction during execution. This permits any SPLP subroutine requiring free storage to obtain it by referencing and then updating that index register, checking to make sure that the top of memory has not been reached.

The closing routine ENDOUT first closes all active files. It then restores the interrupt status to that which existed prior to calling BEGIN, and calls CHNCOM with an argument of 1 to preserve the core image for debugging purposes. It is therefore possible to insert one or more SPLP programs in a macro procedure.

A standard error routine ERRORS is provided. ERRORS determines the type of calling program, and treats calls from MAD differently from FAP calls. From MAD, the call is of the form "ERRORS.($CODE$,-L-)"; $CODE$ is a 6-letter error code and L is an optional argument. ERRORS prints a line on the console containing the code and the location from which the call occurred. If L is present, control then transfers there; if it is absent, control transfers to ENDOUT. FAP calls are handled similarly; if the call to ERRORS was of the form

```
TSX  ERRORS,4
PZE  MMMMM,N-,L-
```

then "NMMMMM" is the error code printed, and control transfers to L or to ENDOUT if L is not present or zero.

The supervisory routines contain two entry points to which transfer is not permitted. The first of these, COMBUF, is a twenty-word buffer which contains the current command buffer at the time the program was called. The other is named REALOC and is a 130-word "workspace" in which string operations are performed. The workspace is treated in many respects as a data item, and can generally be operated on by any SPLP subroutine except the sorting operations as if it were a data item. Neither COMBUF nor the workspace should ever be referenced directly from an SPLP main program.

B. Input-Output Routines.

Subroutines are available in SPLP for inputting and outputting strings to and from either line-marked disk files or the console. Lines from the console can be read by READØ into the workspace or into available storage as data items, and either the workspace or an arbitrary sequence of data items can be written on the console by WRITEØ. The inputs and outputs from these two subroutines are in 12-bit mode, and the console read routine recognizes the standard 12-bit erase and kill characters '#' and '@'.

The disk input-output routines use double buffering for read operations and triple buffering for write operations. Up to three files may be open simultaneously for reading and three for writing. The workspace may either be read to or written from, and sequences of data items can be written in files. Lines cannot be read from disk files directly to free storage.

The files read and written are of line-marked format and must have been opened by calls to one of six opening routines. ROPEN1, ROPEN2, and ROPEN3 open files for reading, first checking to determine whether a file has previously been opened by the same call. If so, the file is closed if it is still open and a check is made to see if the new file to be opened for reading exists. If the new file exists, it is opened for reading; if it does not exist, then a new file first name is requested from the console and the process repeated until a satisfactory name is obtained. The first time that each particular opening program is called, two buffers are obtained from free storage, and every file opened by that program at any later time is assigned the same buffers. Lines from files opened by ROPEN1, ROPEN2, and ROPEN3 are read successively into the workspace by calls to READ1, READ2, and READ3, respectively. Note that it is never necessary to explicitly close a file.

The subroutines concerned with file writing are similar in many respects to their reading counterparts. WOPEN1, WOPEN2, and WOPEN3 open files for writing and assign three buffers to each. The files to be written are checked for nonexistence, and if a file already exists, the user is asked whether or not the file should be deleted. If the answer is yes, the file is deleted and writing may proceed; anything else results in a request for a new file name and a repetition of the nonexistence check. Only permanent, temporary, or secondary mode files can be deleted. In other respects, WOPEN1, WOPEN2, and WOPEN3 are similar to ROPEN1, ROPEN2, and ROPEN3. WRITE1, WRITE2, and WRITE3 write either the workspace or a variable-length sequence of data items in the appropriate disk file, and are similar to WRITEØ, the console writing subroutine, in other respects.

C. Field Finder Routines.

Subroutines may be written in FAP to parse lines in the workspace into data items. Few general rules can be given which apply to these subroutines. The context sensitivity requirement of the previous chapter insures that the parsing can be done if the context is available, but it may be that the required context is not explicitly present, since the workspace contains only one line of input. Assuming that the workspace has been parsed and that a data item has been generated, the problem of inserting the data item in its list in the correct order remains. These problems can be quite difficult, and deserve and require a programmer's best efforts.

Certain conventions have been established for field finders relating to communication with the rest of SPLP. Field finders must return data items, and lists of data must be structured in specific ways. In addition, it is convenient to pass an argument to each field finder which specifies a location to be transferred to in the event that the line could not be parsed.

The list of data must be structured in the general fashion shown in Figure 12. An additional machine word of information not shown in that figure must be included at the left of each element of the list. This element is called a "sort link," and is used in the sorting process. Figure 14 shows a three element list as if it were written in FAP; the FAP instructions of the figure must of course be generated by the field finder. A data item is shown in the box to the left of the figure. Note the twelve-bit strings in the BCI statements.

```
                            PZE    *,,Ø          (SORT LINK)
                       B    PZE    C,,2          (MIDDLE OF LIST)
                            BCI    1,ØD1A1T      (DATA)
                            BCI    1,1A          (DATA)
 ___________
|PZE   B,,A|                  •
 (DATA ITEM)                  •
                              •

                            PZE    *,,Ø          (SORT LINK)
                       A    PZE    B,,3          (TOP OF LIST)
                            BCI    1,ØM101R      (DATA)
                            BCI    1,1EØ 1D      (DATA)
                            BCI    1,1A1T1A      (DATA)

                              •
                              •
                              •

                            PZE    *,,Ø          (SORT LINK)
                       C    PZE    Ø,,1          (BOTTOM OF LIST)
                            BCI    1,ØE1N1D      (DATA)
```

Figure 14. List of data in assembly language.

The data item shown in Figure 14 points to the top of the list (A) and to the element just inserted (B). A points to B, B points to C, and C contains a zero pointer indicating the end of the list. The number of machine words of data in each element is stored in the decrement of the words A, B, and C. The sort links point to themselves.

The situation of Figure 14 might have resulted from a call to a field finder named ABC. Suppose the call to the field finder subroutine ABC was "ABC.(Z,L)". The data item depicted in the figure would have been returned in Z if the field finder had found an appropriate string in the workspace, extracted the string 'Data' from it, and placed the string at B in the list. If the string in the workspace was not appropriate, i.e., not parsable, the field finder would have returned to L and Z would have contained zero.

More than one data item could be returned from a single field finder. If some data items were found and others not found by such a multiple field finder, those data items not found could be returned as zeroes. If ABC were such a multiple field finder and took three arguments, the call might be "ABC.(Z1,Z2, Z3,L)". The calling conventions for field finders need not rigidly adhered to, and are merely introduced as guidelines. The inclusion of the false return label is highly recommended, however.

Included in the programming example of Appendix B is an example of a field finder written in FAP. A careful study of the program should reveal its important features. Recall that the next available storage location is contained in complement form in index register 7.

D. Sorting Routines.

The sorting process was described in Section B of Chapter 3. The algorithm used is somewhat unusual and deserves further comment. The sort is essentially of the bucket variety, and moves pointers to create the buckets which are then merged in preparation for the next pass.

Consider a list of n-tuples of data items and a list of data as it might have been generated by a field finder. During the bucket generation phase of the sort, the pointers from the data items in a particular position in each n-tuple are followed to the list element. The address of each n-tuple is stored indirectly in the decrement and then directly in the address of the sort link associated with the list element pointed to. The first time a sort link is so modified, both the decrement and the address of the link contain the address of the first n-tuple. Thereafter, the last n-tuple looked at is made to point to the current n-tuple and the current n-tuple address is placed in the sort link. The result is a series of lists of n-tuples; the top and bottom of each list is pointed to by some sort link in the list of data. When the bottom of the list of n-tuples has been reached, the top of the list of data is accessed and the merging phase begins. The decrement of the first sort link becomes the pointer to the new list of n-tuples; thereafter, the decrement of each sort link is stored indirectly in the address of the previous sort link until the bottom of the list is reached, at which time a zero is stored indirectly in the address of the last link. The merge phase results in a new list of n-tuples sorted according to the order of the list of data. The cycle is repeated on each position of the n-tuples until the sort is complete.

Figure 15 symbolically summarizes the operations involved. The pairs of boxes represent machine words. The letters represent pointers, and the dashes represent information not used in the sort. Each of the parts of Figure 15 represents one stage in the sort algorithm, and the circled letters indicate the changes at each step.

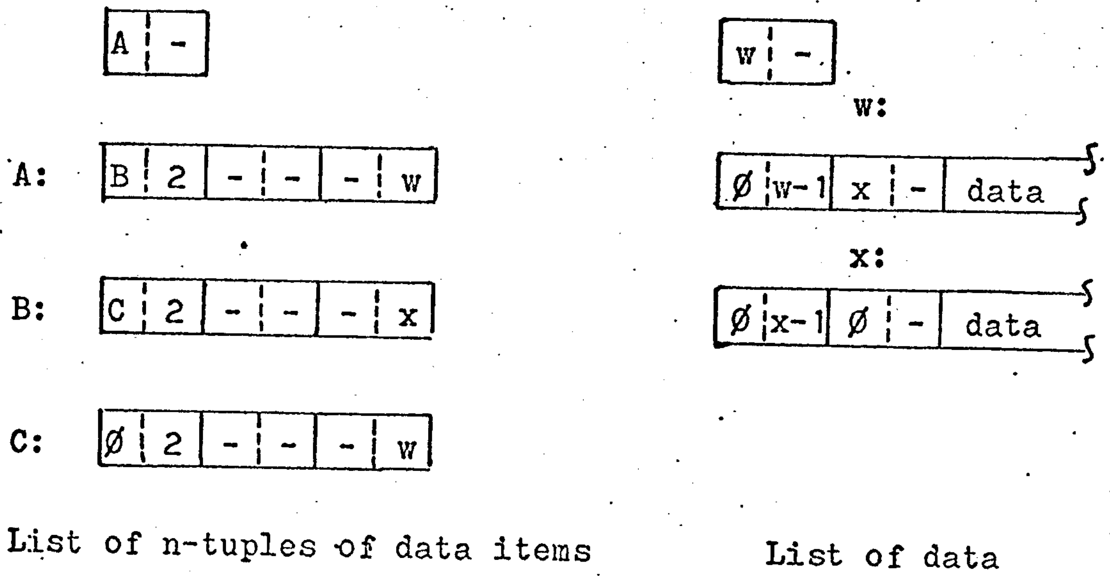
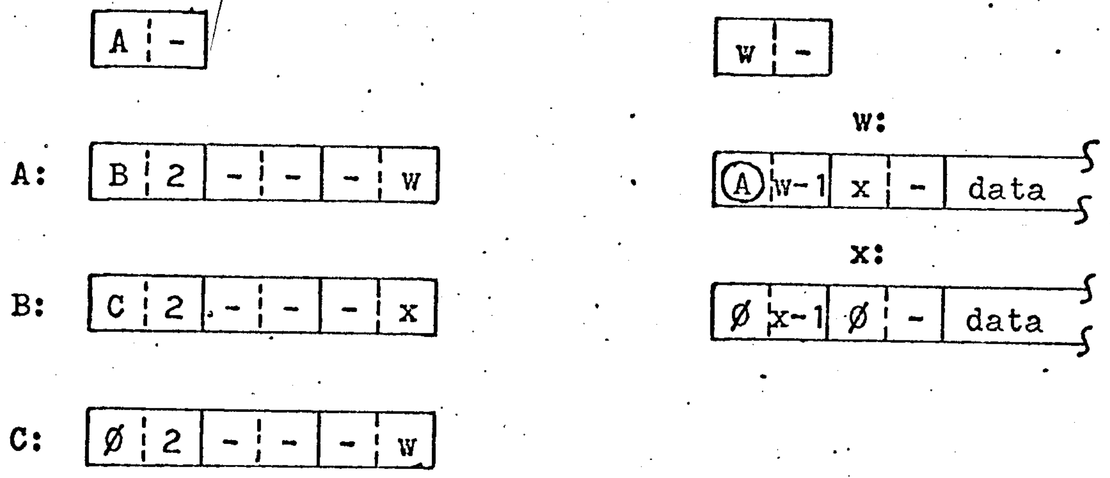
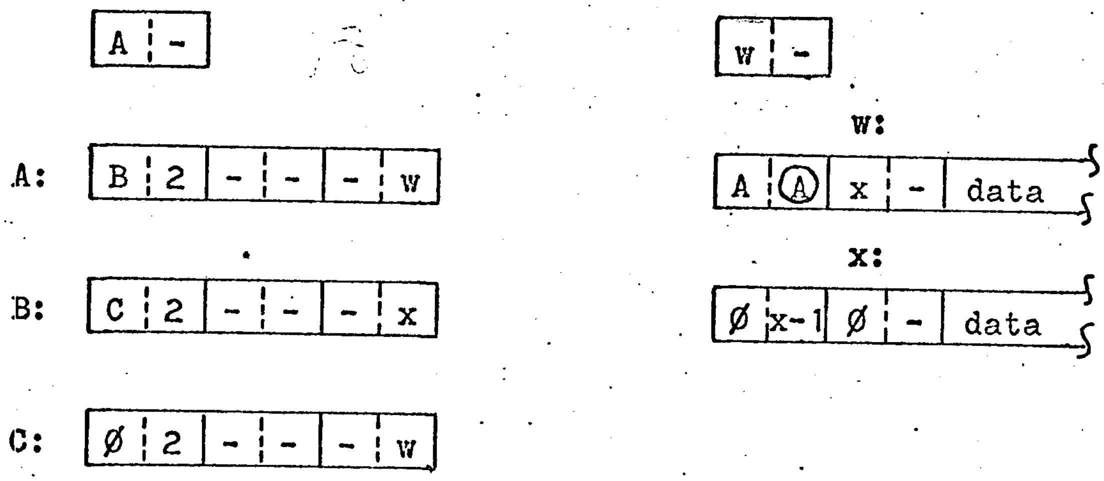


Figure 15a. Before sorting



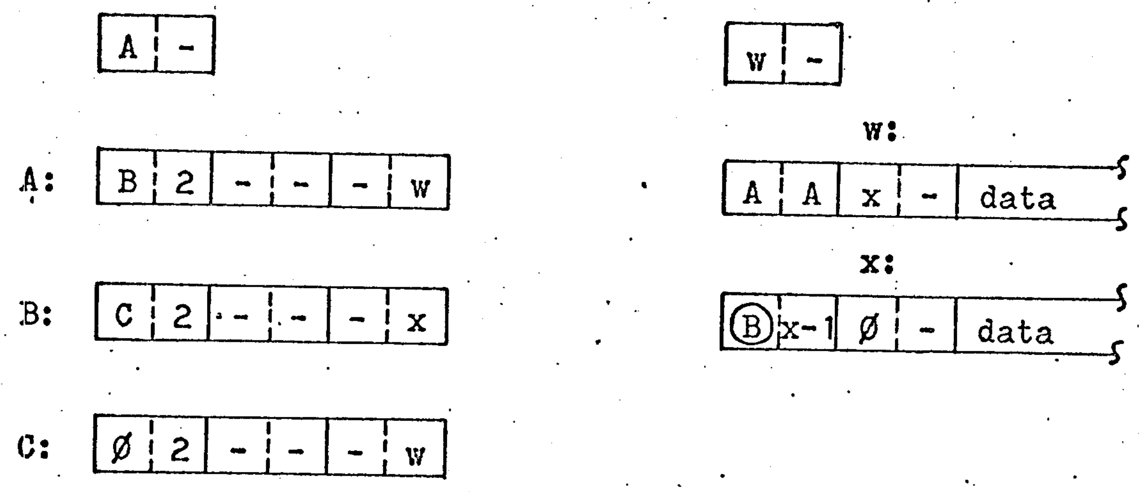
(Store A indirect in the decrement of w's sort link.)

Figure 15b. Step one.



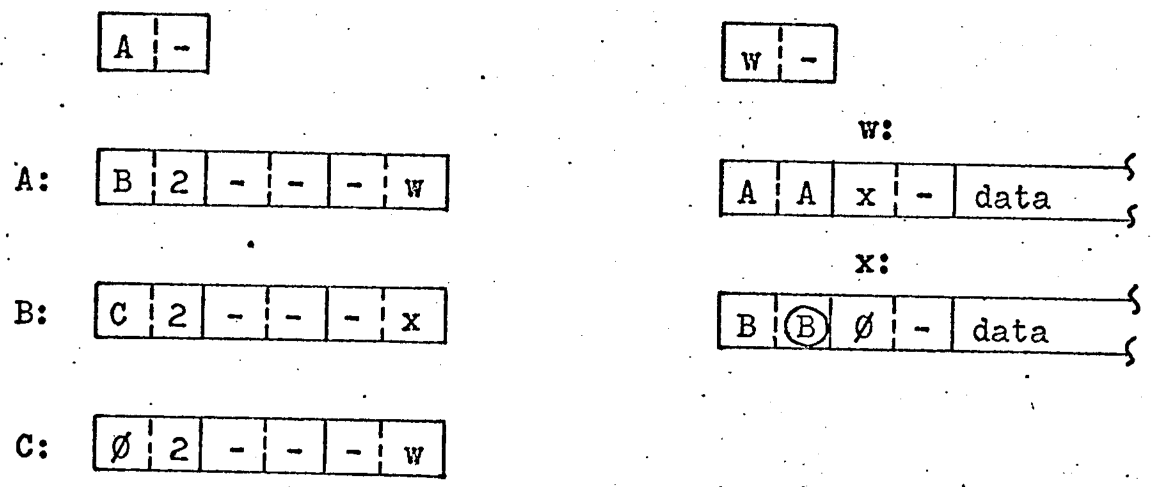
(Store A direct in the address of w's sort link.)

Figure 15c. Step two.



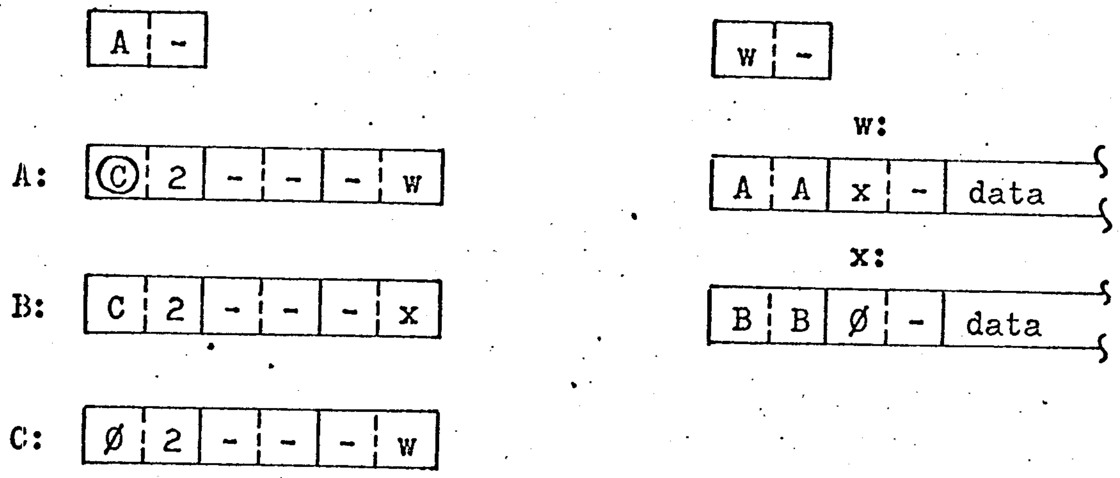
(Store B indirect in the decrement of x's sort link.)

Figure 15d. Step three.



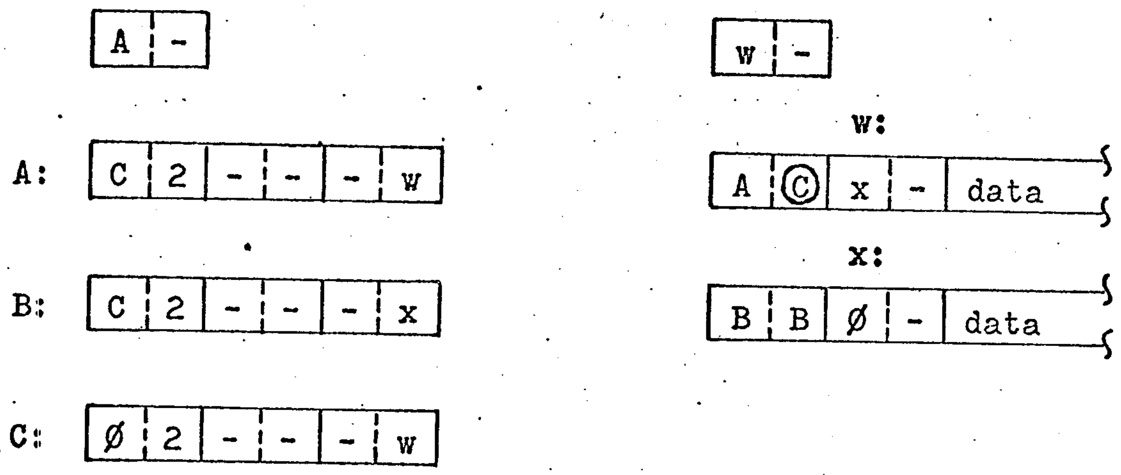
(Store B direct in the address of x's sort link.)

Figure 15e. Step four.



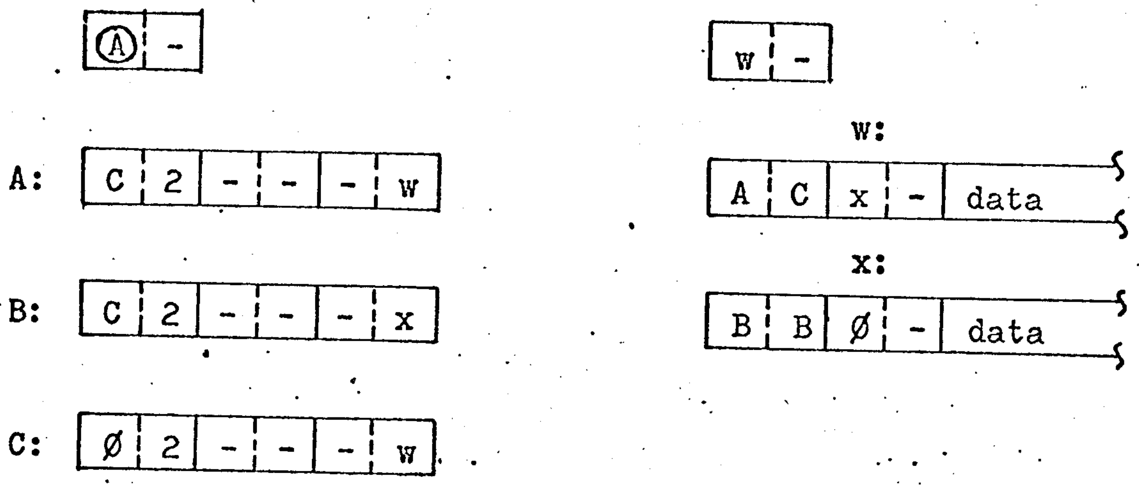
(Store C indirect in the decrement of w's sort link.)

Figure 15f. Step five.



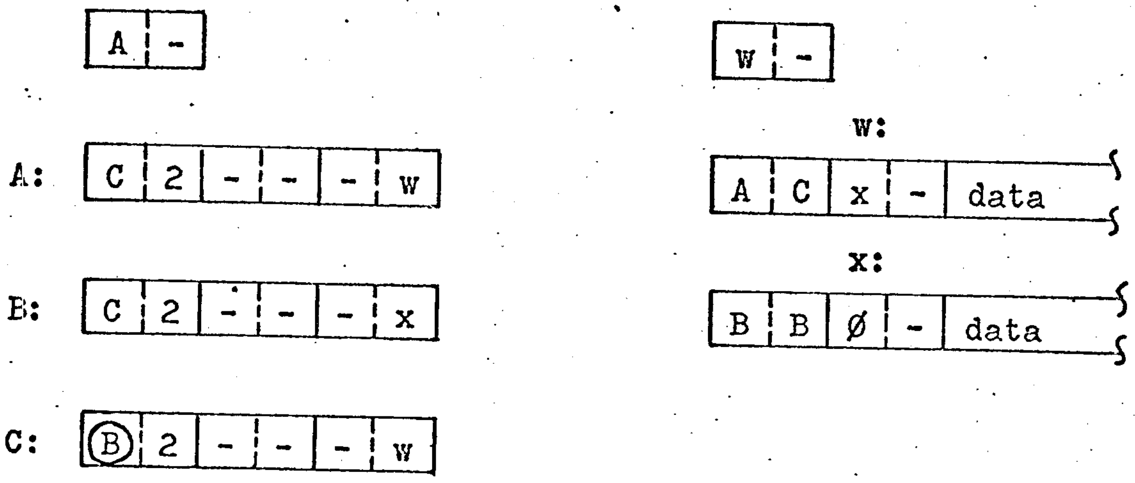
(Store C direct in the address of w's sort link.)

Figure 15g. Step six.



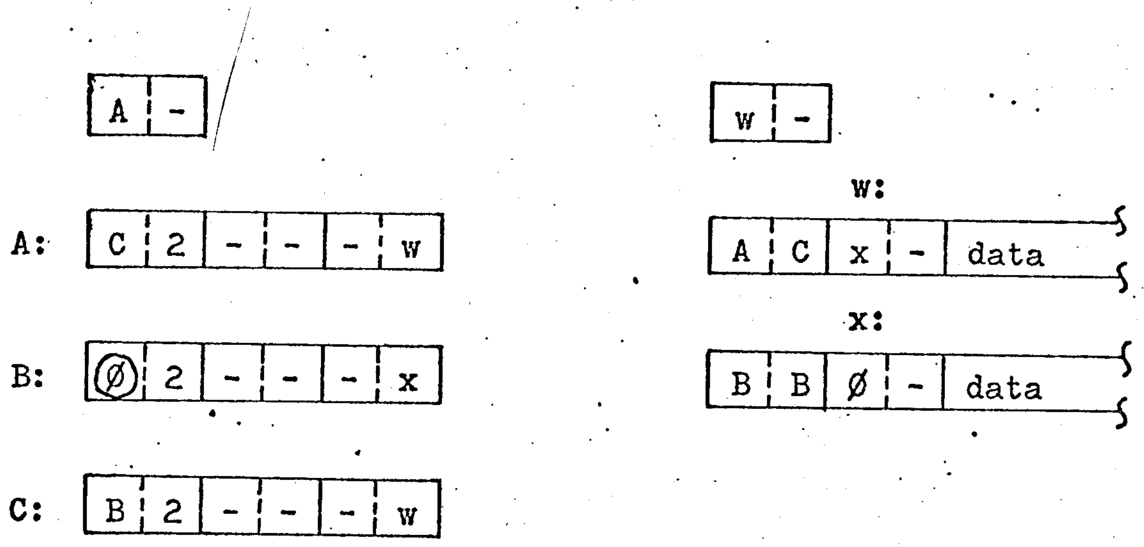
(Store decrement of w's sort link in top of list.)

Figure 15h. Step seven, begin merge.



(Store decrement of x's sort link indirect in w's sort link.)

Figure 15i. Step eight.



(Store zero indirect in x's sort link.)

Figure 15j. Step nine, sort complete.

E. Miscellaneous Routines.

Two subroutines in addition to those described above presently exist in the language. The first of these, QUOTE, generates a data item from its arguments each time it is called. A string from MAD can thus be converted into a data item. The second subroutine, EQUALS, determines whether two data items are equal. Other subroutines will be needed to implement arithmetic, string processing, and list processing capabilities, and are discussed in Appendix A and in the concluding chapter.

CHAPTER 5

CONCLUSIONS

The SPLP language as discussed above is incomplete in a number of respects. Facilities for arithmetic, string processing, and list operations are planned. Arithmetic operations will probably be accomplished in MAD, with conversion routines provided in SPLP to transform strings to and from integer or real numbers. String processing of various kinds can be performed in the workspace, and if a powerful enough capability is available, the parsing operations involved in extracting the data from the input strings can be largely performed by the string processor. What is needed is a versatile string substitution subroutine. The list operations needed in SPLP are several. It should be possible to create lists, append to them, insert elements, and so forth. Erasure should not be needed. It should also be possible to search lists for the existence of data items. These facilities should be easily incorporated.

Often it is desirable to "lock" data for privacy. String enciphering and deciphering routines are planned for SPLP; they will permit a user to deny access to certain data by unauthorized users. The access control problem is an important one in any data management context; the problem of controlling access to relations in a data base is still unsolved.

The SPLP language is a relatively powerful instrument for applications involving data streams. The data stream concept is quite general and describes many real information structures. Although SPLP was conceived in response to a particular requirement, the language is not biased toward its current applications, except in the sense that efficiency considerations have dictated that checking be kept to a minimum.

Many problems of a theoretical nature remain unsolved. First, the problem of efficient implementation of a data base in a particular hardware configuration is presently quite intractable. Questions relating to the design of hardware for data management may well be more capable of solution. Perhaps the most efficient kind of machine for data retrieval will employ a very large, slow random access memory. A further question centers around determining a measure of the information content in a data base; an answer to this question is necessary if a meaningful metric of implementation efficiency is to be found. Finally, the problem of parsing and ordering data strings from lightly structured textual data streams is important; a language of some kind is needed to allow specification of the syntax of the strings and an order on each element of the syntax in a concise and meaningful way.

APPENDIX A

SPLP PROGRAMMER'S MANUAL

Listed below are the commands of SPLP. Commands marked with an asterisk are not yet implemented. Argument types are denoted symbolically as follows:

Z: Data item
L: Statement label variable
N: Integer variable
F: Real variable
B: Boolean variable
P: Password

It is strongly suggested that the MAD declaration "NORMAL MODE IS INTEGER" be inserted in SPLP programs to prevent unexpected mode conversion. No SPLP subroutines recognize MAD "block notation"; the notation "Z1,Z2,...ZN" which appears below signifies that the argument list is variable in length but must be specified at compile time.

BEGIN.

Must be the first executable function in any SPLP program. BEGIN extends memory bound, sets one level of interrupt, and saves the current command buffer. If '(DBUG)' is in the current command buffer, BEGIN calls the loader to load FAPDBG.

ENDOUT.

Closes all active files, resets one level of interrupt, and calls the supervisor via CHNCOM.

ERRORS.($MESS$,-L-)

Prints an error message of the form 'ERRORS CALLED FROM ABSOLUTE LOCATION (LOC), CODE MESS'. Only the first six characters of MESS are printed. ERRORS then transfers control to L if present, otherwise to ENDOUT. Numeric codes are used for calls internal to SPLP and should be avoided.

READO.(Z,L)

Causes a line from the console to be placed in Z. If Z=-0, the line is placed in the workspace. If not, a new data item Z is created in free storage containing the inputted line. If an empty line (i.e., a carriage return) is typed, control transfers to L.

WRITEO.(N,Z1,Z2,...ZN)

Causes data in Z1 through ZN to be written on the console. If N=0, the workspace is written.

ROPEN1.($NAME1$,$NAME2$)

ROPEN2.($NAME1$,$NAME2$)

ROPEN3.($NAME1$,$NAME2$)

Open files for reading. If $NAME1$=0, it will be requested; if $NAME2$=0, it will be assumed $(MEMO)$. Files opened by previous calls will be closed, so that a maximum of three files may be read simultaneously. If a file to be opened for reading does not exist, a new file first name will be requested.

READ1.(L)

READ2.(L)

READ3.(L)

Read one line from the line-marked file last opened for reading by ROPEN1, ROPEN2, or ROPEN3 respectively. The line is put in the workspace. End-of file results in a transfer to L.

WOPEN1.($NAME1$,$NAME2$)

WOPEN2.($NAME1$,$NAME2$)

WOPEN3.($NAME1$,$NAME2$)

Open files for writing. If $NAME1$=0, it will be requested; if $NAME2$=0, it will be assumed $(MEMO)$. Files opened by previous calls will be closed, so that a maximum of three files may be written simultaneously. If a file to be opened for writing already exists, a request to delete it will be made. If the request is granted, the file will be deleted; if the request is denied, a new file first name will be requested.

WRITE1.(N,Z1,Z2,...ZN)

WRITE2.(N,Z1,Z2,...ZN)

WRITE3.(N,Z1,Z2,...ZN)

Cause data in Z1 through ZN to be written in line-marked format as a single line in the file last opened for writing by WOPEN1, WOPEN2, or WOPEN3, respectively. If N=0, the workspace is written.

*GET.(Z,L)

Places the workspace in Z. If the workspace is empty, control is transferred to L.

*PUT.(N,Z1,Z2,...ZN)

Places Z1 through ZN in the workspace. If N=0, the workspace is cleared.

*SAVE.(N)

Saves the workspace in a temporary buffer numbered N (N .LE. 7).

*RESTOR.(N)

Places the contents of the Nth temporary buffer in the workspace (N .LE. 7).

*B=SCAN.(Z1,Z2,L)

Scans Z1 for an occurence of Z2. If Z1 or Z2 =-0, the workspace is meant. If an occurence of Z2 is not found in Z1, control transfers to L and B will be false. If an occurence is found, B will be true and a normal return will be taken. If L=0, a normal return will be taken in any event.

B=EQUALS.(Z1,Z2,L)

Checks for equality between Z1 and Z2. If Z1 or Z2 =-0, the workspace is meant. If the two are not equal, control transfers to L and B will be false. If the two are equal, B will be true and a normal return will be taken. If L=0, a normal return will be taken in any event.

*B=CHANGE.(Z1,Z2,Z3,L)

Changes the first of the longest occurences of Z2 in the workspace to Z3. The first occurence of Z1 in Z2 stands for zero or more characters in the workspace and may or may not appear in Z3. If Z1 does not occur in Z2, it is ignored and stands for itself in Z3. If Z1=0, it is ignored. If no occurence of Z2 in the workspace is found, control transfers to L and B will be false. In this case, no substitution takes place. If an occurence is found, Z3 replaces Z2 in the workspace and a normal return is taken with B true. If L=0, a normal return will be taken in any event.

A few examples might be helpful. In the following, ƀ stands for blank. The contents of Z1, Z2, and Z3 are given, and the workspace is shown before and after the command.

```
Z1: $
Z2: $,ƀ
Z3:
Before: Jones,ƀA.ƀC.
After: A.ƀC.

Z1: $
Z2: $
Z3: $ƀ
Before: A.ƀC.
After: A.ƀC.ƀ

Z1: $
Z2: ,$
Z3:
Before: Jones,ƀA.ƀC.
After: Jones

Z1:
Z2:
Z3: A.ƀC.ƀ
Before: Jones
After: A.ƀC.ƀJones
```

If D contains '$', DCB contains '$,ƀ', DB contains '$ƀ', and CD contains ',$', the above command sequence can be implemented as

```
SAVE.(1)
CHANGE.(D,DCB,0,ERR)
CHANGE.(D,D,DB,ERR)
GET.(FIRST,ERR)
RESTOR.(1)
CHANGE.(D,CD,0,ERR)
CHANGE.(0,0,FIRST,ERR)
```

FINDER.(Z1,Z2,...Zn)

Is a field finder of n arguments. The workspace is parsed, and n data items Z1, Z2, ...Zn are created in free storage and inserted in the appropriate lists. If any Zi=-0, that data item is not created in free storage. If the workspace cannot be parsed successfully, control transfers to L.

IN.(N,Z1,Z2,...ZN)

Generates a vector containing the data specified by the Zi. All calls to IN must have the same number of arguments and hence the same N, until a call to OUT results in a normal return.

SORT.(N,Z1,Z2,...ZN)

Sorts the vectors generated by successive calls to IN in lexicographic order. N and the number of arguments must agree with IN. The sort only takes place if the elements of the vectors generated by IN are the result of list generation functions and if the arguments of IN are in fixed order. Otherwise, the result of SORT is undefined.

OUT.(N,L1,L2,...LN)

Outputs the vectors sorted by SORT to the N arguments of SORT, one at a time. Successive vectors are outputted via repeated calls to OUT. Control is transferred to the L which corresponds to the leftmost argument of SORT which changed since the last call to OUT. When no vectors remain to be outputted, a normal return is taken, and IN may be called with a new N.

*Z=COMBIN.(N,Z1,Z2,...ZN)

Combines the lists of data items specified by the Zi into a single list Z. The data item referred to by Z is the first data item in the list Z1.

*Z2=NEXT.(Z1,L)

Obtains the data item following Z1 on the list indicated by Z1. If Z1 is the last item on its list, Z2=0 and control transfers to L.

Z=QUOTE.($0T1W1E1L1V1E0-1B1I1T0 1S1T1R1I1N1G$)

Creates a new data item in free storage containing the argument to QUOTE.

*N=INT.(Z,L)

Converts the data item Z into an integer. If Z is not a number, control transfers to L.

*F=REAL.(Z,L)

Converts the data item Z into a real number. If Z is not a number, control transfers to L.

*Z=STINT.(N)

Creates a new data item Z in free storage containing the string which represents the integer N.

*Z=STREAL.(F)

Creates a new data item in free storage containing the string which represents the real number F.

*N1=COMAND.(N,L)

Returns the Nth argument of the current command buffer when BEGIN was called in the integer (hollerith) variable N1. If the command buffer has fewer than N arguments, control transfers to L, otherwise the normal return is taken.

*P=KEY.(Z)

Converts Z into a password suitable for use by ENCRYP and DECRYP. Z may be any length. P should not be modified and is not suitable for printing.

*ENCRYP.(P)

Enciphers the workspace with password P. The result is suitable for printing. Up to three million characters may be enciphered with the same P.

*DECRYP.(P,L)

Deciphers the workspace with password P. Correctness of the password is not checked internally; however, the format of the workspace is checked. An enciphered line has a special character format for all but the last six characters, which must be integers. If the format is not correct for an enciphered line, DECIPH will transfer control to L. DECIPH uses the integers to maintain sync; a transposition of two enciphered lines will result in an error.

APPENDIX B

PROGRAMMING EXAMPLE

A. Field Finder.

Consider a class of strings such that an object called <u>salary</u> is always expressed by the first six characters of each string in the class, and a string is in the class if and only if the first six characters of the string are numerals. A field finder written in FAP to recognize and order these objects in decreasing order is given below. The call is SALARY.(Z,L).

```
        ENTRY SALARY
SALARY  SXA   XR2,2           SAVE XR2
        LXC   $REALOC,2       WORKSPACE LOC.
        CAL   1,2             FIRST 3 CH.
        LDQ   2,2             SECOND 3 CH.
        PAI
        OSI   2,2             'OR' CH.
        OFT   =0776077607760  CHECK NUMERIC
        TRA   FALSE           NOT NUMERIC
        CAL*  1,4             DATA ITEM...
        TNZ   *+2
        TMI   XR2             NOT NEEDED.
        AXT   SALIST,2        TOP OF LIST
        SXA   *+1,2
ORDER   LDC   **,2            GET NEXT LIST ELT.
        TXL   PUT,2,0         END OF LIST
        LAS   1,2             FIRST 3 CH:
        TRA   PUT             GREATER, STORE
        TRA   CHECK2          CHECK NEXT 3
        SCA   ORDER,2         LESS, GO ON
        TRA   ORDER
CHECK2  XCL                   NEXT 3 IN AC.
        LAS   2,2             NEXT 3 CH:
        TRA   REV             GREATER, STORE
        TRA   EQUAL           EQUAL, DON'T STORE
        XCL                   LESS, GO ON
        SCA   ORDER,2
        TRA   ORDER
REV     XCL                   SWAP BACK
PUT     TXL   MEM,7,4         TEST FOR ROOM
```

```
                SLW    2,7                  STORE CHARS.
                STQ    3,7
                PCA    0,7
                SLW    0,7                  SORT LINK
                TNX    MEM,7,1              ALLOCATE 1 WORD
                PCD    0,2                  NEW LIST PTR.
                ADD    =2                   WORD COUNT
                SLW    0,7                  DATA COMPLETE
                PCD    0,7
                AXT    0,2
                STD*   ORDER                MODIFY OLD PTR:
                SCA    SALIST,7             MODIFY DATA ITEM
                TNX    MEM,7,3              ALLOCATE 3 WDS.
                TRA    *+2
EQUAL           SCA    SALIST,2             MODIFY DATA ITEM
                CAL    SALIST
                SLW*   1,4                  RETURN DATA ITEM
XR2             AXT    **,2                 RESTORE XR2
                TRA    3,4                  NORMAL RETURN
*
FALSE           ZAC
                LXA    XR2,2                RESTORE XR2
                TRA*   2,4                  FALSE RETURN
*
SALIST          PZE    *
*
MEM             TSX    $ERRORS,4            FATAL ERROR-
                PZE    0,2                  MEMORY EXHAUSTED.
                END
```

B. SPLP Program.

Consider now a data stream containing personnel information which is structured as follows:

1. One or more names;
2. For each name, a salary;
3. For each name, one or more "fringe benefits".

Let the field finders NAME.(Z,L), SALARY.(Z,L), and FRINGE.(Z,L) exist and be capable of parsing three different kinds of input lines, one for each kind of personnel information. We wish to invert the data stream to obtain a list of fringe benefits, salaries, and names as follows: