AD709875

RM-6265/3-PR
JUNE 1970

THE MIND SYSTEM: THE STRUCTURE OF THE SEMANTIC FILE

Martin Kay and Stanley Y. W. Su

D D C
RECEIVED
AUG 14 1970
C

prepared for

UNITED STATES AIR FORCE PROJECT RAND

Rand
SANTA MONICA, CA. 90406

Best Available Copy

2

RM-6265/3-PR
JUNE 1970

THE MIND SYSTEM: THE STRUCTURE OF THE SEMANTIC FILE

Martin Kay and Stanley Y. W. Su

[redacted]

This research is supported by the United States Air Force under Project RAND—Contract No. F44620-67-C-0045—Monitored by the Direcorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of the United States Air Force.

Wash DC 20330

Rand
SANTA MONICA, CA. 90406

THIS DOCUMENT HAS BEEN APPROVED FOR PUBLIC RELEASE AND SALE; ITS DISTRIBUTION IS UNLIMITED

DOCUMENT CONTROL DATA

1. ORIGINATING ACTIVITY The Rand Corporation	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP
3. REPORT TITLE THE MIND SYSTEM: THE STRUCTURE OF THE SEMANTIC FILE	
4. AUTHOR(S) (Last name, first name, initial) Kay, Martin, Stanley Y. W. Su	
5. REPORT DATE June 1970	6a. TOTAL NO. OF PAGES 71 — 6b. NO. OF REFS. ---
7. CONTRACT OR GRANT NO. F44620-67-C-0045	8. ORIGINATOR'S REPORT NO. RM-6265/3-PR
9a. AVAILABILITY/LIMITATION NOTICES DDC-1	9b. SPONSORING AGENCY United States Air Force Project RAND

10. ABSTRACT

One of a series of memoranda describing the design, implementation, and use of the MIND system, ~~this~~ study provides a detailed description of the way in which the main fact file of the MIND information-management system is organized. The format of each type of record is given in the form of a declaration in the PL/1 programming language. A method is described for organizing long lists of items so that any desired entry on the list can be retrieved with a minimum of accesses (usually one) to the disk. Also included is a set of computer procedures which can be incorporated into any program that has cause to refer to the file. These make it easy to store and retrieve information and to relieve the programmer of the necessity of bearing all the details of the file representation constantly in mind.

11. KEY WORDS

Linguistics
MIND (Question-Answering System)
Information Systems
Semantics

PREFACE

This is one of a series of papers describing the design, implementation, and use of the MIND* system. The design goals for the MIND system are responsive to the increasingly urgent need for a means of fast and accurate information transactions between relatively senior command, control, and policymaking personnel, on the one hand, and very large, heterogeneous, loosely-formatted information banks on the other. The system is an unobtrusive servant; it understands, acts upon, and replies with, English sentences; its users will require no special training. It is thus a prototype for a class of systems that are well suited to the critical task of unifying, controlling, and exploiting the massive and often chaotic flow of information that centers upon the senior command levels of the Air Force and other services, especially in emergency situations.

The MIND system consists of nested and chained modules of high level programming language statements, and it is therefore relatively easy to modify, either for improvement or for adaptation to specialized applications.

*Management of Information through Natural Discourse.

SUMMARY

This paper gives a detailed description of the way in which the main fact file of the MIND system is organized. The format of each type of record is given in the form of a declaration in the PL/1 programming language. A method is described for organizing long lists of items so that any desired entry on the list can be retrieved with a minimum of accesses (usually one) to the disk.

A set of computer procedures is described which can be incorporated into any program that has cause to refer to the file. These make it easy to store and retrieve information and to relieve the programmer of the necessity of bearing all the details of the file representation constantly in mind.

CONTENTS

THE MIND SYSTEM: THE STRUCTURE OF THE SEMANTIC FILE

1. INTRODUCTION

This is one of a series of reports on the design of the MIND* information-management system presently under way in the RAND Linguistics Project. An information-management program, as we understand it, should be capable of reading text in an ordinary language, say English, and responding to questions on the information that it contains in the same language. Such a program must clearly contain numerous routines for extracting information from ordinary text and for restating it in some canonical format. There must be routines for inserting new information in the central file of facts, and for retrieving the information needed to construct answers to particular questions. In this paper we are concerned with the central file of facts itself and with the ways in which the program will be able to organize data in it.

In Section 2 we shall be concerned with the ways in which individual items of information will be physically arranged on IBM 2314 disks. The reader will be assumed to have some knowledge of the grosser properties of this kind of storage device.

*Management of Information through Natural Discourse

Section 3 describes a set of procedures provided in the MIND system for manipulating the file—entering new and retrieving old information. Eight of these procedures are used by the program that constitutes the semantic component of the MIND system; the remainder perform more elementary functions and are intended mainly for use by the first eight. The chart in section 4 illustrates the structure of the complete file—manipulation package showing which procedures make use of which others.

All the procedures mentioned here are written in the PL/1 programming language. Some familiarity with PL/1 will be useful, though not essential, for understanding what follows.

The file itself consists of a complex network of *items* that can enter into a number of different relations. If two items, *a* and *b*, enter into a relation *R*, then the record representing *a* will contain, among other things, the address of *b* labeled with the name of the relation, *R*. The record corresponding to *b* will contain the address of *a* labeled with the name of the converse relation, *R*'. Given any item in the file, it is therefore possible to find the location of any other item to which it is related.

It is a property of many of the relations we are concerned with that if a pair of items, *a* and *b*, are in the relation, then there is no *c* different from *b* such that _a_ and _c_ are in the relation. Notice that this property can hold for a relation without holding for its converse. We shall refer to a relation for which the property holds as a _singular_ relation; all others are _multiple_ relations. As we shall see, there is good reason for providing slightly different physical representations for the two kinds of relation.

The main aim of the design set out in this paper has been to provide a file structure that will make it as easy as possible to search for particular items, items that stand in specified relationships to other items, to collect together sets of items that share specified properties, to perform the usual operations of set theory upon such sets, and to do all these things with few references to the disk so that the amount of time required will not become unreasonably great.

The file consists, as we have said, of a collection of items which can enter into relations of various kinds. An item will be represented in the file by a set of one or more _blocks_. Blocks are of two kinds. A _label_ block consists of a list, each entry of which contains the name of a relation and the address of some other block in the file. If the relation named is singular, then the entry contains the address of the label block corresponding to an item that stands in the specified relationship to the present one. If the relation is multiple, then the entry contains the address of a link block, and the link block contains the addresses of the other items related to the present one in the way specified.

Figure 1 shows an example of how these different kinds of blocks might be used. "Grass" and "green" are connected through a relation called "color". The label block for "grass" therefore contains an entry labeled "color" and the address of the "green" label block. Since an object or substance only has one color, the "color" relation is singular. However, since various objects and substances can have the same color, the converse of the "color" relation is multiple. The name of a given relation cannot appear more than once in a label block. The entry in the "green" label block corresponding to the relationship "converse color" therefore points to a link block containing the addresses of all green items in the file. This is intended simply to illustrate the way in which relations are stored in the file and not to show how the system would translate the English sentence "Grass is green." For reasons beyond the scope of this paper, color would probably be represented by a label block and not by a relation. The translation of the sentence would be considerably more complex.

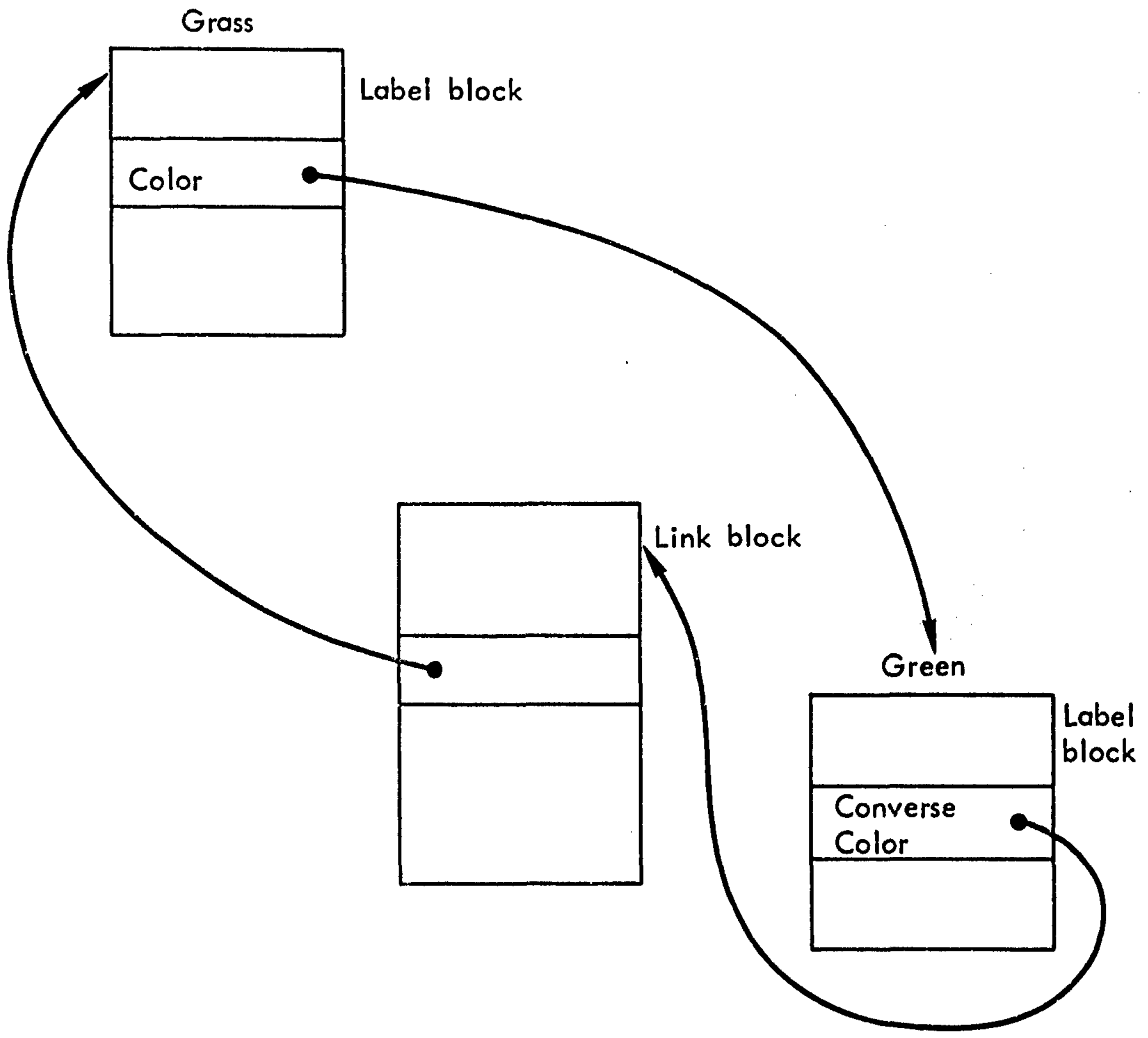


Fig. 1—The internal representation of a relation and its converse

2. FILE STRUCTURE

2.1. Data Tracks

Physically, the central file is a regional data set on disk which consists of a number of records. For reasons of operating efficiency a disk track is treated as a logical record. The following PL/1 declaration shows the structure of a data track.

```
DCL 1 TRACK BASED(QTK),
      2 LABELDRT(254) BIT(13),
      2 LINKDRT(254) BIT(13),
      2 AVLABEL BIT(13),
      2 AVLINK BIT(13),
      2 COLLECT BIT(10),
      2 LBCOUNT BIT(8),
      2 LKCOUNT BIT(8),
      2 BLOCKS(6462) CHAR(1);
```

Each data track contains (1) two directories specifying the addresses of the label blocks and link blocks in the track, (2) two available space pointers(AVLABEL and AVLINK) giving the addresses of the first location available for label and link blocks respectively, (3) an accumulator(COLLECT) which keeps track of the number of items deleted from the track, and controls the operation of garbage collection, (4) two counters(LBCOUNT and LKCOUNT) which specify the block numbers of the first available label block and link block respectively, and (5) the main body of the track, which takes the form of an array of characters (BLOCKS) in which label blocks and link blocks are situated.

2.1.1 Label Blocks The detailed structure of a label block is shown in the following PL/1 declaration:

```
DCL 1 LABEL BASED(Q),
      2 LENGTH BIT(8),
      2 CONTINUE BIT(8),
      2 LINKS(500),
        3 NAME BIT(7),
        3 MULTIPLE BIT(1),
        3 ITEMAD,
          4 DUMMY BIT(1),
          4 MEASURE BIT(8),
          4 LINKAD,
            5 TRACK BIT(15),
            5 OFFSET BIT(8);
```

A label block may contain a maximum of 500 named links. The first 8-bit field(LENGTH) gives the number of named links in the block. The 8-bit field(CONTINUE) is provided in the block structure to refer to a continuation block on the same data track. If a continuation block has to be put on a different track, this field contains the value 255('11111111'B) and a different mechanism (to be described in Section 2.2) is used to locate the continuation block.

Each names link contains (1) a 7-bit field for the name of a relation, (2) one bit(MULTIPLL) to show whether the link points to a label block or a link block, (3) an 8-bit field(MEASURE) which contains a measure. (See below) There is one unused bit labeled DUMMY. (4) a 15-bit field(TRACK) which contains the address of a track in which the referred block is to be found, and (5) an 8-bit offset(OFFSET) giving the position of the block in that track. The links in label blocks are sorted based on the value in the first 7-bit field. If the bit(MULTIPLE) is zero, the link points to a label block, i.e. the address of an item in the file. Otherwise the link points to a link block. In the latter case, the relation has to be multiple.

2.1.2 Link Blocks. The structure of a link block is as follows:

```
DCL 1 LINK BASED(Q1),
      2 LENGTH BIT(8),
      2 CONTINUE BIT(8),
      2 LINKS(500),
        3 DUMMY BIT(1),
        3 MEASURE BIT(8),
        3 LINKAD,
          4 TRACK BIT(15),
          4 OFFSET BIT(8);
```

A link block may contain a maximum of 500 links. An 8-bit field gives the length, i.e. the number of links that the block contains. A second 8-bit field may contain, as in the case of the label block, the reference to another link Each link consists of an 8-bit measure, one unused bit, and a reference to a label block (the address of an item) in the form of a 15-bit track address and an 8-bit offset. item) in form of a 15-bit track address and an 8-bit offset.

The links in a list of link blocks are arranged in descending order according to internal 31-bit values (8-bit measure, 15-bit track number and 8-bit offset). The value of the 8-bit measure will be determined in such a way that those links which are most likely to be used would appear in the earlier portion of the list. One way to do this is to give a higher measure to blocks representing more general facts and a lower measure to more specific facts. The generality of a proposition might be measured by counting the number of quantifiers it contains.

2.2. Continuation Blocks

2.2.1. Overall Strategy. The file is a complex network of label and link blocks. From time to time it may be necessary to add new entries to an existing label or link block which, in general, belongs to a list of blocks. The system can store the new entries in four possible ways. First, it can store the new entries in an existing block if there is room for them. Second, the new entries may be stored in the block before or after the current block in the list. Third, if there is no space available in the adjacent blocks, the system may obtain a new block from the space available in the current track to accommodate the new entry, and insert it in the list. The number of spaces in each new label or link block will always be some definite amount which will be variable in the light of experience. Tentatively, the figure is set at 4 for both label and link blocks. If a new block is established with only two links, then space for two additional entries in the block will remain unused. However, if two additional entries have to be added to the block at some later time, the cost of doing so will be relatively small. Fourth, the system can increase the block size of an existing block to make room for new entries.

It is clearly desirable to increase the size of a particular block rather than to establish a new block as a continuation of the original one. One reason is that this will save the cost of establishing a new block, i.e. the space occupied by the continuation pointer and the length of a block. Another, and more important, reason is that, since an 8-bit offset is used to address blocks in a data track, the system can only establish a maximum of 254 label blocks and 254 link blocks. Increasing the block size would, therefore, allow the system to use more space in a track. However, we must consider the cost of rearranging the material on the original track to obtain space which is physically consecutive with the original block.

Two parameters of the system, variable in the light of experience, are used to determine whether a continuation block will be established on the same track as the block that it continues, or whether the material on the track will be rearranged so that the original block can be made larger. Let us assume that label blocks are established at one end of the space on the track and link blocks at the other so that the space available for new blocks is somewhere in the middle. The amount of labor involved in increasing the size of an existing block depends on the number of blocks intervening between it and the available space in the center of the track. If this number exceeds a particular value established in advance, then a continuation block will be established. Otherwise, the current block will be extended and the blocks intervening between it and the available space in the center of the track will be moved to make room. The advantage of establishing label and link blocks at opposite ends of the track lies precisely in the fact that the expected amount of material intervening between a given block and the space available for new blocks is greatly reduced.

The value established in advance will be different for the two kinds of blocks so that the policy governing the allocation of new space also regulates the proportion of the space in any given track made available to label and link blocks because, in this way, it is possible to increase the likelihood that the link blocks corresponding to a given label block will be on the same track and, therefore, to reduce the time expected to access all the information related to a given item.

It is obviously desirable that the removal to other positions in a track of the intervening blocks between a given block and the available space should not entail changes to references to this block from other places in the file. Each track on the disk will therefore contain a pair of <u>directories</u> with 254 13-bit entries each. One directory is used to refer to all the link blocks in the track, and the other for label blocks. An offset refers directly to a position in one of these directories and the entry at that position points to the block. It is the use of these directories that makes it possible to refer to the position of the given block within a track with only 8 bits. 13 bits makes it possible to refer to 8,192 different actual locations in the track which is more than adequate since each track on the IBM 2314 disk pack contains 7,294 addressable locations that can be used for data.

From time to time, blocks may be deleted from the file, thus producing pockets of unused space at unexpected places on a track. When the available space in the center of the track is used up, the first move is therefore to carry out the procedure that has come to be known as <u>garbage collection</u>. If blocks in the track have been deleted (the 10-bit field COLLECT specifies the number of deletions made), then the material on the track is rearranged so as to bring the available space toward the center of the track, between that occupied by the label and the link blocks. Only when no further space can be reclaimed in this way is a new track brought into service. As a result of deletions, space may become available on a track which was previously completely occupied. Blocks on that track that have continuation blocks on some other track make first claim on that space. Whenever convenient, continuation blocks are moved onto the track that contains the blocks they continue so that they can be referred to more cheaply whenever the original block is referred to. The opportunity to perform a maneuver of this kind cheaply can be expected to occur fairly often because, if operations on the file require reference to a particular block, then they will typically also require reference to its continuation blocks. The information on the tracks containing the two kinds of blocks can therefore be expected to be in the rapid access store of the computer at the same time in the normal course of events. What is required is, therefore, that the program should be on the lookout for this situation and that it should move continuation blocks onto the same track as the block they continue whenever the two tracks on which they currently reside happen to be in the computer together.

Even if both tracks are available at the same time, it is probably desirable to move a block from one track to another only when a certain minimum amount of space, possibly more than that required for the block to be moved, is available on the track. If the block were moved whenever there was just sufficient space to accommodate it, then there is every reason to suppose that the number of links in that block would shortly be increased so that a new continuation block would again have to be established on a different track. In this case, little would have been gained in making the move.

2.2.2. The Track Continuation Directory. Even given the strategy of moving continuations back to the original track just described, lists of label blocks or link blocks may at times extend from one track to several other tracks. Information concerning the blocks that continue, and the positions of the continuation blocks, needs to be recorded. One way to do this is to store this information in the same track as the lists of label or link blocks that are continued. However, there is a serious drawback to doing so, namely that it may be necessary to read many tracks from the disk into core storage just to find the position of the desired continuation block. It is clearly desirable to avoid all unnecessary disk accesses and to go directly to the track where a link should be or has been stored. A directory is therefore established which contains continuation information for all tracks in the file. When storing or retrieving a piece of information in the file, the system first refers to this directory to determine the track in which the information should be. To avoid ambiguity, we shall, in the following discussion, call the tracks which are occupied by the directory the _directory tracks_ and those which contain link and label blocks the _data tracks_.

Each entry of the track continuation directory contains the continuation information for one data track in the file. The track continuation information in an entry is stored in a list of fixed-length blocks. The following PL/1 declaration shows the structure of the directory track.

```
DCL 1 TKCONTINUE BASED(R),
      2 TBLOCKS(68),
        3 NEXTBK BIT(8),
        3 ELEMENTS(12),
          4 INFOR1,
            5 OROFFSET BIT(8),
            5 RELATION BIT(7),
          4 LAST,
          5 DUMMY BIT(1),
          5 MEASURE BIT(8),
          5 TERM,
            6 LASTTK# BIT(15),
            6 LASTOFFSET BIT(8),
        4 INOFR2,
          5 CONTRACK# BIT(15),
          5 CONOFFSET BIT(8),
    2 AVL,
      3 AVLPOINTER BIN FIXED(15),
      3 AVLCOUNT BIN FIXED(15),
    2 CONTINUE,
      3 TK# BIT(15),
      3 OFFSET BIT(8);
```

A directory track contains 68 fixed-length blocks, each of which contains the following information: the first 8-bit field refers to a continuation block in the same track, if there is one; otherwise all eight bits are set to zero. It is followed by 12 elements. All elements in a block and its continuation blocks are ordered according to the internal representation (47-bit value) of the substructures INFOR1 and LAST. Each element contains information concerning the starting point of a list of label or link blocks in a track, say X, which has continuations in a track Y, and also contains information concerning the position of the block in track Y to which the list continues.

If the data in an element concerns the track continuation of a list of link blocks, the 8-bit field OROFFSET gives the offset of the label block (the address of an item), and RELATION gives the name of the relation with which the list of link blocks is associated. These fields are followed by a 32-bit field LAST, which is a copy of the last link in the list of link blocks in track X, and by another 23-bit field INFOR2, which gives the position of the continuation block in track Y.

If the data in an element concerns the continuation of a list of label blocks, the position of the first label block (the address of an item) in a list of label blocks is identified by the 8-bit field OROFFSET, and LASTØFFSET contains the name of the last relation on the list in track X. The location of the continuation block in track Y is given by the 23-bit field named INFOR2. RELATIØN, MEASURE and LASTTK# have the value zero.

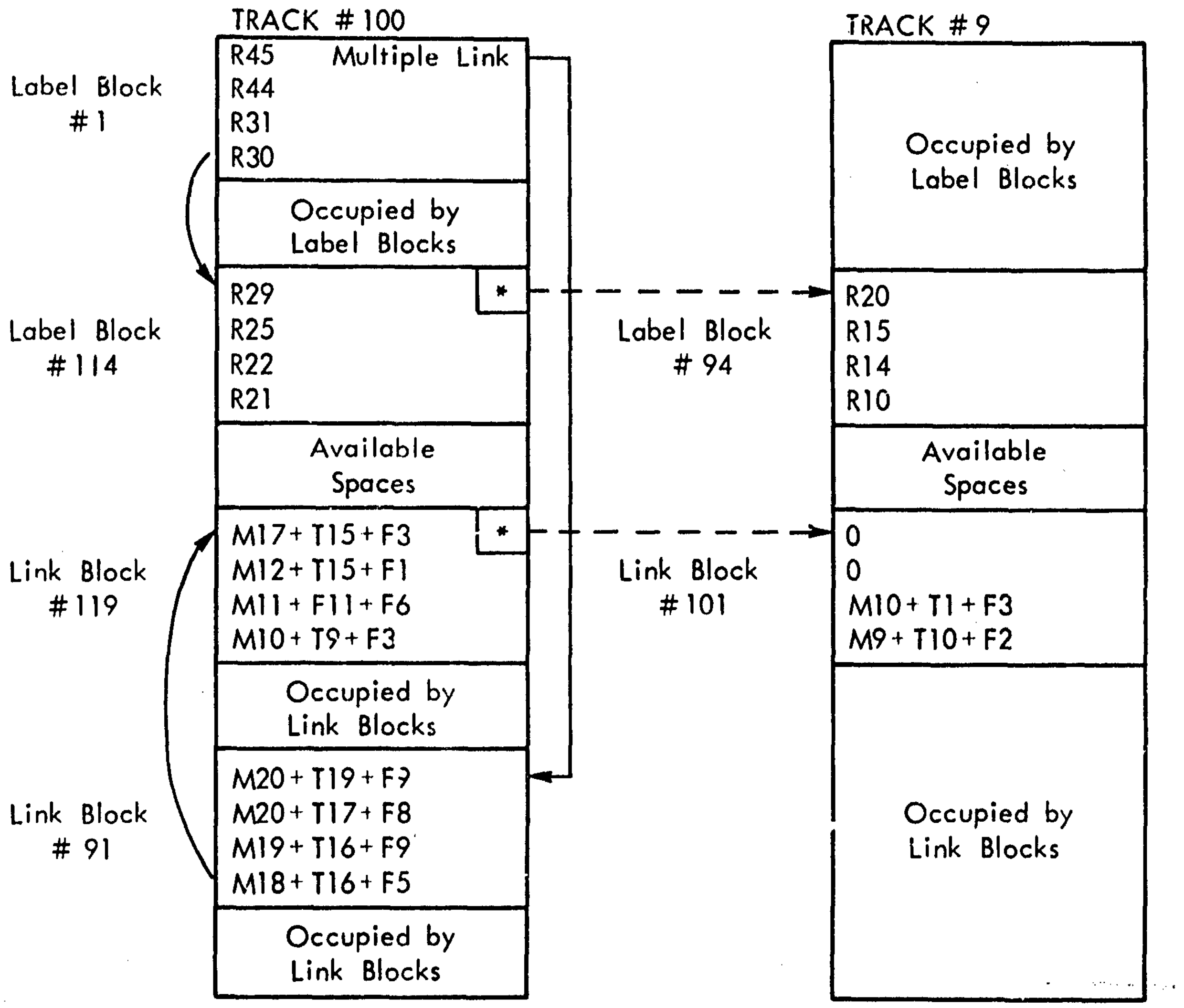
Each directory track contains information concerning the unused space in the track. A pointer AVLPOINTER gives the position of the first available block in the track and a counter AVLCOUNT gives the number of available blocks. Each directory track has a continuation pointer CONTINUE which gives the position of a block in another directory track to which the track continues. The use of this continuation pointer will be described later.

Figure 2 illustrates how the continuation information is stored in the directory when a list of label blocks, and a list of link blocks in track 100 have continuation blocks in track 9.

Figure 2(a), the R's stand for relation names, M's for measures, T's for track numbers, F's for offsets and '*' is a special marker which indicates that a block has continuations in another track. Notice that the links in a list of link blocks are arranged in descending order according to 31-bit internal values consisting of the measure, the track number, and the offset fields that represent a link. The links in a list of label blocks are ordered according to the internal representation of relation names (7-bit values).

The following will illustrate the use of the directory. Let us suppose that an item represented by T100+F1* is to be connected to the item represented by T25+F3 through relation R45. Assuming that the item T25+F3 has measure M9, we want to insert a link M9+T25+F3 in a list of link blocks that starts at the position T100+F1 with relation name R45. We first check the entry in the directory for track number 100 to determine whether any continuation information concerning the item T100+F1 and the

*The '+' sign is used here to represent concatenation.



(a) The Data Tracks

8 OR OFFSET	7 RELATION	9 MEASURE	15 LAST TK #	8 LAST OFFSET	15 CONTRACK #	8 CON OFFSET
1	R45	10	9	3	9	101
1	0	0	0	R21	9	94

(b) Two Directory Elements

Fig.2—An example of lists on more than one track

relation R45 have previously been stored. In our example we would find the first element shown in Figure 2(b). Since, as specified in this element, the last link of the list of blocks in track 100 is M10+T9+F3, whose internal representation is greater than the link M9+T25+F3, M9+T25+F3 should be stored in the continuation block which is in T9+F101. This information allows the system to access track 9 without reading track 100 into core storage. Thus, to store or retrieve a link in a list of blocks, we can obtain the proper track from disk by referring to the directory without having to trace the list through many tracks. The directory allows the system to go directly to the relevant portion of the file and eliminates unnecessary disk accesses.

The track continuation directory may itself occupy several tracks. Its size would depend on the number of lists of label or link blocks in data tracks that have continuation blocks in other tracks. Some heuristics may be applied to help control the size of the directory. For example, we may allow a continuation block to go onto a data track only if the track contains a certain minimum amount of space, so that data on the new track would not shortly be increased and need again be extended to a different track. Or we may reduce the inter-track continuations by consolidating the continuation blocks into a minimal number of tracks as space becomes available to do so. Both of these strategies would reduce the amount of continuation information to be entered into the directory.

If the size of the directory is small, it will be kept in core storage. Nevertheless, as the data in the file continues to grow, the size of the directory may exceed the space that the system can spare. Part of the directory may have to be stored on a disk. If we allow portions of the continuation information of a data track to spread over several directory tracks (which may at times be on disk), we may have to do several disk accesses just to find a piece of track continuation information. This would completely destroy the advantage of the directory. We therefore move blocks from one directory track to another if necessary to keep all the continuation information associated with each data track in the same directory track. Thus, we can obtain the continuation information of a data track by one disk access at the most if the directory track happens to be on a disk at the time when it is needed. We must also countenance occasional complete reorganizations of the file, copying it onto an entirely new set of tracks and reducing to an absolute minimum the number of continuations from one track to another.

A directory track is generally large enough to hold the continuation information for any data track. A directory track will need a continuation track only in the unusual case where a data track contains a large number of label blocks having labels whose associated multiple links are stored in other tracks. A continuation pointer CONTINUE is provided for each directory track to specify the location where the track continues. However, the need for a continuation track for a directory track is not likely to occur since the method of entering data into a track provides for keeping label blocks and link blocks within a track in proper proportion.

2.3. Resident Data Tables

In the preceding sections we have described the structures and the use of both data and directory tracks. The data tracks are normally stored on disk and are brought into core storage when data on the tracks are to be used or modified. The directory tracks, just as data tracks, can be moved in and out of core storage. However, as a general policy, we keep as many directory tracks in core as possible. In order for the system to access tracks efficiently, some general information concerning the content of the tracks on disk and also of the tracks brought into core storage needs to be kept in core by the system. In this section, we shall describe this information (the data tables) which is to be kept in core for ready reference, and the scheme used to maintain the tracks brought into core storage.

2.3.1. The Master Index. For each data track that contains label or link blocks which have continuation blocks in other tracks there will, as we have seen, be lists of fixed-length blocks in a directory track which contain the information concerning all the continuations. The location of the head blocks and the lengths of these lists are stored in the master index table as shown in Figure 3. The track number of a data track is used as an index to specify the position at which the information is stored. The index of the table starts from zero because tracks in the file are numbered beginning with zero. N is the number of tracks used in the system.

TK#	FLAG	DIRECTORY TRACK	OFFSET	LENGTH	TYPE
0					
1					
2					
⋮	NEWGLAG	CØNTK#	CONTKØF	LENGTH	TKTYPE
.					
.					
.					
N-1					

Fig. 3 — The Master Index

The fields CONTK# and CONTKOF specify the head block of a list in a directory track which contains the track continuation information of a data track. The length of the list (LENGTH) is used to determine which list of blocks is to be moved to other directory tracks in order to keep all continuation information for a given data track, as far as possible, in a same directory track. The shortest list will be moved first when the moving operation is required. The 1—bit flag (NEWFLAG) specifies whether a new item or a continuation block can be established in the associated data track. When the available space in a data track is below a threshold predetermined by the system, this bit is set to zero and no new item or continuation block will be established in the track. The 1—bit field TKTYPE specifies whether the associated track is a data track or a directory track. For directory tracks and for data tracks which do not contain any items that have continuation blocks in other tracks, the associated fields CONTKOF and LENGTH on the table will be set to zero and CONTK# will be set to (15)'1'B.

2.3.2 The Resident—Track Directory. We shall now describe the scheme employed in the system to manage the tracks which are at times brought into core storage. First let us consider the space in core which is to be reserved as working space for manipulating data in the file. The working area is large enough for M tracks, where M is a system parameter whose value will be determined on the basis of the core storage remaining after space for programs in the system has been allocated. During the process of data manipulation, data tracks and directory tracks will be moved in or out of this working space. When a data track or a directory track is to be brought into core from disk and no space in the working area is available, a track in the working area must either be moved out to disk or erased to make room for the incoming track. To maintain some kind of priority among the tracks currently in the working space, a queue is established on the basis of how recently the tracks have been used. When a track is used, it is positioned at the head of the queue. A track at the tail of the queue will be removed or erased when space in the working area is needed.

The system will maintain a table shown in Figure 4(a) to specify the tracks that are currently in core and show whether the data in the tracks has been modified. When the content of a track has been changed (i.e., data inserted or deleted), the status flag will be set accordingly. The track can then either be written out onto disk or simply erased in order to make room for another track. The example in Figure 4(a) shows that tracks 2, 5, 4 and 15 are currently in core and the data in tracks 2 and 4 have been modified.

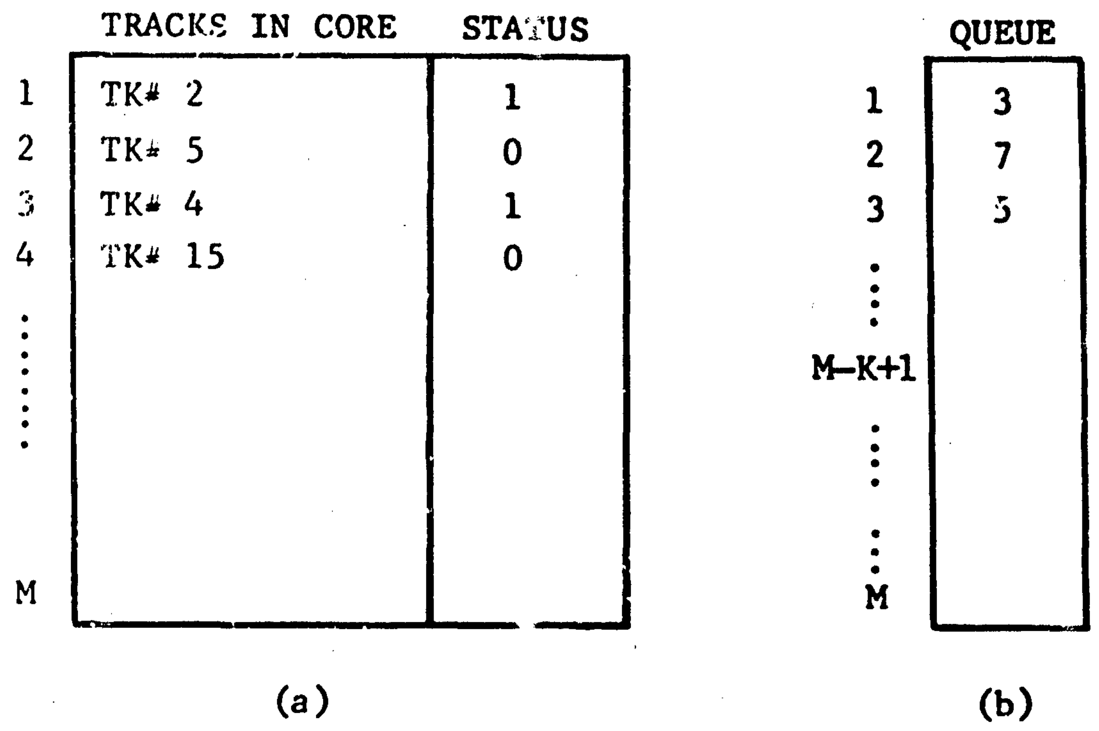


Fig. 4 — The Resident—Track Directory

Figure 4(b) shows a queue which is an array of pointers. The pointers specify the positions in the table (Figure 4(a)) at which the track number and status are stored. In the above example, 3 is on top of the queue. It specifies the position of track number 4 in Figure 4(a). Thus, track number 4 has been used most recently. Whenever a data track or a directory track is used, the system re—arranges the pointers in the queue to maintain proper priority among the tracks in the working space.

Since the same working space is used for both data tracks and directory tracks, it may contain any number of these two types of tracks at a given time. As new tracks are brought into the working space, old tracks, generally from the bottom of the queue, must be either written out or erased. Whenever possible, data tracks are released from core before directory tracks for the following reason: Since the system first refers to directory tracks in order to determine which data tracks contain the relevant data, it is advantageous to retain as many directory tracks as possible in core. As described in the preceding section, the information in directory tracks allows the system to go directly to the relevant portion of the data in the file. If the size of the directory is small, all the directory tracks will be kept in core for continuous reference; otherwise, they will be moved in and out of the working space. In order to assign to directory tracks a higher priority for remaining in core than data tracks, we introduce another system parameter, k, where $1<k<M$ as shown in Figure 4(b). When a track is to be moved out from the working space to make room for an incoming track, the k pointers at the bottom of the queue are searched (bottom up). A data track, specified by a pointer, rather than a directory track, will be moved out to disk or erased, regardless of the incoming track type. A directory track is to be removed from the working space only when none of the k pointers at the bottom of the queue specifies a data track.

2.4. Core Lists

In order to provide for temporary lists which will be used by various procedures and then discarded, there is an in core list processing system which will, nevertheless, allow a list in core to have a list on disk as a sublist.

The declaration for the core list space is:

```
DCL  1  LSTSPCE EXTERNAL,
        2 HEAD(2) BIT(15),
        2 TAIL(2) BIT(15),
        2 ELT(20000) CHAR(6);
```

HEAD(1) and TAIL(1) point to the beginning and end respectively of the available space list of single elements, while HEAD(2) and TAIL(2) point to the available space list of double elements. Single and double elements are distinguishable by the contents of an ID field (see below). Initially all the available elements are tied together into the available space list of double elements. The subscript of the ELT array, which identifies the location of a single element or of the first element of a double element, will hereafter be referred to as the _index_ of that single or double element. A double element will always consist of two consecutive members of the ELT array.

Every list element contains a 2 bit ID field with the following values:

'00'B — a single element which contains data.

'01'B — a single element which contains the index of a core list header or the address of a disk list.

'10'B — a double element which contains the indices of the first and last elements of a core list and the length of that list.

'11'B — a double element which serves as a reader of a core or disk list.

These elements are referred to as a _data element_, a _list name_, a _header_ and a _reader_ respectively.

2.4.1. Data Elements. The declaration of a data element is:

```
DCL  1 ELEM BASED(P),
       2 ID BIT(2),
       2 NEXT BIT(15),
       2 DATA BIT(31);
```

The NEXT field contains the index of the next element in the same core list as the element itself. This field contains all zeroes for the last element in a list.

When a data element contains the internal name of an item, the DATA field is structured as follows:

```
       2 MEASURE BIT(8),
       2 TRACK BIT(15),
       2 OFFSET BIT(8),
```

If the offset is all zeroes, the item is in the core file, otherwise it is in the disk file.

<u>2.4.2. List Names</u>. A list name has the following structure:

```
DCL  1 LSTNME BASED(PN),
       2 LID BIT(2),
       2 LNEXT BIT(15),
       2 LHDR BIT(15),
       2 LOFFSET BIT(8),
       2 LABEL BIT(8);
```

When this element is the name of a core list, LHDR contains the index of the header of the list and LOFFSET and LABEL contain all zeroes. When it is the name of a disk list, LHDR and LOFFSET contain the track and offset respectively of the item from which the list starts, and LABEL contains the label which identifies the pointer list.

<u>2.4.3. Header</u>. A header has the following structure:

```
DCL  1 HEADER BASED(PH),
       2 HID BIT(2),
       2 FIRST BIT(15),
       2 LAST BIT(15),
       2 LENGTH BIT(15),
       2 MARK BIT(1);
```

FIRST contains the index of the first element of the list or zeroes if the list is empty. LAST contains the index of the last element of the list or the index of the header itself if the list is empty. LENGTH contains the length of the list. MARK is used in the internal operation of some of the procedures.

<u>2.4.4. Readers</u>. A <u>reader</u> functions like a pointer or bookmark. A reader is always associated with a list and is, at any given moment, pointing at one of the elements of that list. Readers can be created and destroyed at will and are associated in the act of creation with the list they will be used to read. Any number of readers can be associated with a given list and the lists can be either on disk or in core. Furthermore, lists of readers can be created and other readers associated with these lists. The system provides procedures for creating new readers and also for causing an existing reader to point to a new element of its list. A particular case of this latter operation, known as <u>incrementing</u> the reader, causes it to point to the next element following the one it currently points to.

The structure of a reader is:

```
1 READER BASED(Q),
  2 RID BIT(2),
  2 RNEXT BIT(15),
  2 TYPE BIT(1),
  2 RDATA BIT(78);
```

TYPE is '0'B if the reader is reading a core list and '1B' if it is reading a list on disk. RDATA has a slightly different internal structure depending on whether the list being read is in core or on disk. In the former case, it contains the addresses of the current element and the one just preceding it on the list; in the latter case, the pointer to the preceding element is replaced by a pointer to the entry in the track-continuation directory showing the further tracks, if any, on which the list is continued.

3. ACCESS PROCEDURES

This section describes a set of programs—procedures written in the PL/1 programming language—that facilitate the construction of other programs using a file with the kind of structure described in the preceding sections. These procedures can be thought of as extensions of the PL/1 programming language itself. They carry out operations which, though sometimes complex, can most conveniently be viewed as elementary by the writer of a larger program. Thus, for example, the semantic component of the MIND system uses these procedures rather than manipulating the semantic file directly. The writer of this program was thereby relieved of the necessity of managing continuation tracks, priority lists and the like. His job was simplified, the resulting program was more perspicuous, and a situation was produced in which details of the file structure can be changed without redesigning the main semantic program.

There are eight principal operations made available to the programmer through these procedures. They enable him to

(1) Establish a new label block in the file,

(2) Locate the item or items standing in a specified relation to a given item,

(3) Establish a relation between a pair of items,

(4) Create a reader,

(5) Move a reader so that it points to a new item,

(6) Get the data associated with a reader,

(7) Verify if a specified relation is associated with a given item,

(8) Terminate operations on the file, returning all information in core to disk storage.

These procedures are described in subsection 3.1 below. Some of the procedures are fairly complex and some of the work that is done in one must also be done in others. It has therefore proved profitable to create some yet more elementary procedures for use in constructing these. The writer of a high—level program, say a semantic component for the MIND system, is not expected to require direct access to these procedures except on rare occasions. However, no cost attaches to making them available to him and they are therefore described; those used to manipulate material on data tracks are listed in subsection 3.2 and those that affect directory tracks in subsection 3.3.

3.1. Storage and Retrieval Procedures

This section describes eight procedures used in the semantic component of the MIND system for the storage and retrieval of information in the semantic file.

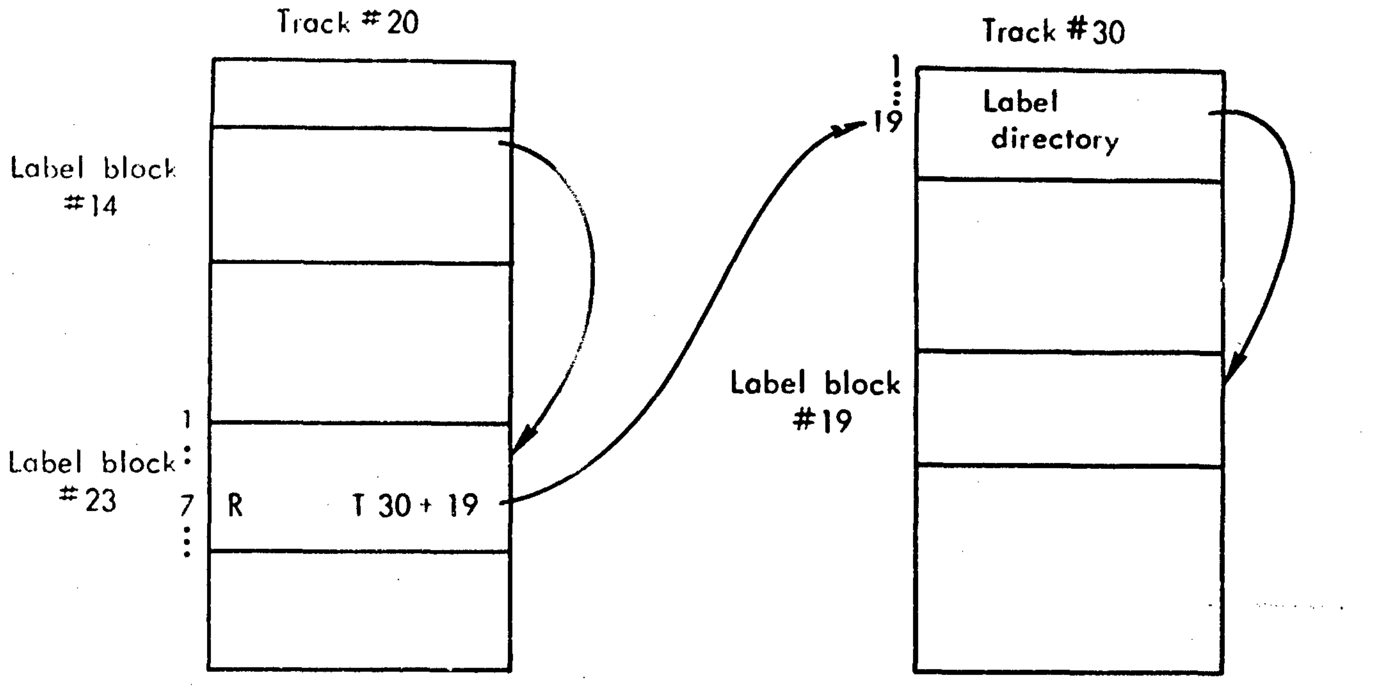
3.1.1. NEWITEM BIT(23).

This function establishes a new label block in the file and returns its location as the value of the function. The first 15 bits of the value returned gives the number of the track where the new block is situated and the remaining 8 bits give the index of the entry in the label directory of that track where its actual location (offset) on the track is stored. Space for the new block is sought (1) in a data track currently residing in the core store of the computer; if no such space is available, then (2) in some other track that forms part of the current semantic file; if this fails, then (3) in a new track which is now incorporated into the file for the first time.

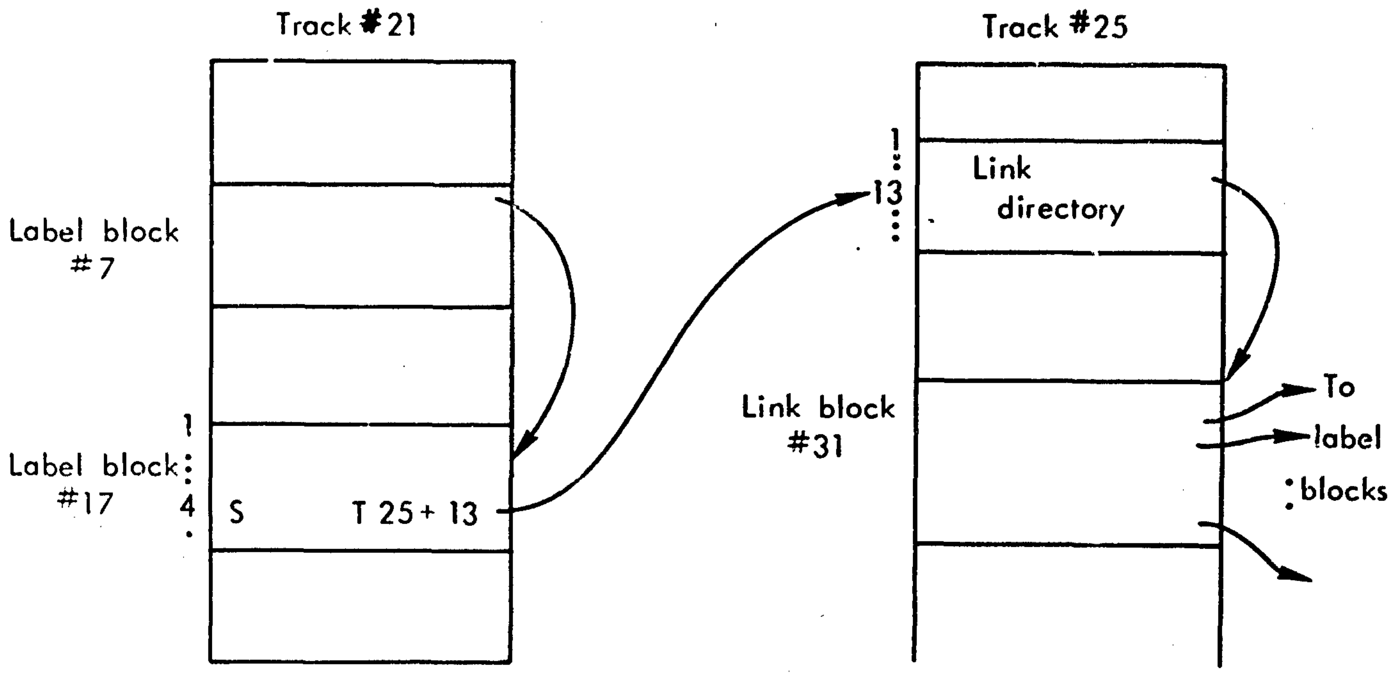
3.1.2. RELFOL(RL,ITEM,ANS) BIT(1). The declarations of the parameters are as follows:

```
        DCL  RL BIT(7),
             1 ITEM,
               2 ITEM_TRACK BIT(15),
               2 ITEM_BLOCK# BIT(8),
             1 ANS,
               2 ANS_TRACK BIT(15),
               2 ANS_BLOCK# BIT(8),
               2 ANS_INDEX BIT(8);
```

ITEM gives the address of a label block a and RL the name of a relation R. If R is a singular relation, then the function attempts to locate another label block b such that a R b holds. If such a label block exists in the file, then ANS will contain, on return, the address of that block. ANS_INDEX will contain the value 0. If R is a multiple relation, then the function attempts to locate the list of link blocks each of whose entries contains the address of a label block b such that a R b holds. If such a list



(a) Singular Relation



(b) Multiple Relation

Fig.5—The RELFOL Procedure

exists, then ANS will contain, on return, the address of the label-block entry that contains the address of that list. In either case, the value of the function is '1'B if a suitable block is found in the file, and '0'B otherwise.

Consider the examples depicted in Fig. 5. If ITEM has the value T20+23 and the relation R is singular, then the value of the function will be '1'B and ANS will have the value T30+19+0. If ITEM has the value T21+7, the relation S is multiple, and the list of label blocks headed by the one at T21+7 has an entry for S in the 4th position of the block at T21+17, then the function will have the value '1'B and ANS will contain T21+17+4.

3.1.3. CONNECT(LITEM,RL,RITEM). The declarations of the parameters are as follows:

```
DCL  1 LITEM,
        2 LITEM_TRACK BIT(15),
        2 LITEM_BLOCK# BIT(8),
     1 RL,
        2 RL_NAME BIT(7),
        2 RL_MULT BIT(1),
     1 RITEM,
        2 RITEM_MEASURE BIT(8),
        2 RITEM_TRACK BIT(15),
        2 RITEM_BLOCK# BIT(8);
```

RL specifies a relation which the CONNECT procedure will establish between the items named in LITEM and RITEM respectively. In other words, if LITEM contains the address of a label block <u>l</u>, RITEM the address of a label block <u>r</u> and RL the name of a relation R, then CONNECT will put the proposition <u>l</u> <u>R</u> <u>r</u> into the file.

<u>3.1.4. SETRDR(L,D) BIT(15)</u> The declarations of the parameters are as follows:

```
DCL 1 L,
      2 L_TRACK BIT(15),
      2 L_BLOCK# BIT(8),
      2 L_INDEX BIT(8),
    1 D BIT(31);
```

SETRDR creates a new reader (see 2.4.4) and returns, as the value of the function, a 15-bit string* called a <u>reader index</u>* which characterizes it and which will be used to identify it to other procedures. L specifies the list to be read, that is, the list with which the new reader is to be associated. D is either all zero or contains a reference to a label block in the form typically stored in a link block.
block in the form typically stored in a link block. If
If L_BLOCK# and L_INDEX are zero, then L_TRACK will be taken

*A reader index is, in fact, an unsigned integer which is used to index an external array called ELT. If <u>i</u> is a reader index, then ELT(i) is the address of the associated reader.

as the address of a list in core; otherwise L will be understood as the address of an entry in a label block containing the name of a multiple relation and, therefore, the address of a list of link blocks. In any case, L specifies the list with which the reader is to be associated. D is either all zero, or it contains a so-called <u>data item</u>. If D is zero, then the reader will be established pointing to the first item on the list referred to by L; otherwise the reader will point to the first entry on the list with a data value less than or equal to D.

Typically, readers will be used to scan lists of link blocks. Recall that these contain references to label blocks stored in descending order. A reference to a label block consists, for these purposes, of an 8-bit measure, a 15-bit track number and an 8-bit offset concatenated together. D will normally contain such a 31-bit quantity.

If D contains a quantity less than any on the list, so that the reader cannot be set up as directed, then no reader is created and zero is returned as the value of the function.

<u>3.1.5. INCRDR(R,D) BIT(1)</u> The declarations of the parameters are as follows:

```
DCL  R BIT(15),
     D BIT(31);
```

R is a reader index and D is a data item (see 3.1.4). INCRDR is used to cause the reader identified by R to point to a new entry on its associated list. If D is zero, then the new entry will be the one immediately following the current one. Otherwise the reader is moved down the list until an entry is encountered with a data value less than or equal to D. If such an entry is found, then the reader is left pointing to that item; otherwise the RTHIS field of the reader is set to zero, the RLAST field points to the last entry on the list, and '0'B is returned as the value of the function. If the reader does not run off the end of the list and is left pointing at an entry, then the value of the function will be '1'B.

3.1.6. RDRDATA(RINDEX) BIT(31) The declaration of the parameter is as follows:

```
DCL  RINDEX BIT(15);
```

RDRDATA returns the datum of the element specified by the first pointer of a reader. In the case where the reader is associated with a list of link blocks, this datum will be a 31-bit string consisting of an 8-bit measure followed by a 15-bit track number and an 8-bit offset. RINDEX is the reader index.

3.1.7. LØCRL (ITEM,RL) BIT(1) The declarations of the parameters are as follows:

```
DCL  1 ITEM,
       2 TK# BIT(15),
       2 ØF BIT(8),
1 RL,
  2 NAME BIT(7),
  2 MUL BIT(1);
```

This function searches for a relation name RL in a label block or a list of label blocks representing the item ITEM. If the relation name is found, '1'B is returned. Otherwise, '0'B is returned.

<u>3.1.8. QUIT</u> This procedure does the final book-keeping before the system terminates its operation on the semantic file. It stores away the system variables and dumps those tracks in core, whose contents have been modified, to the data set on disc.

3.2. Data-Track Procedures

The procedures to be described below carry out the following operations: (1) locate an item which enters into a given relation with another item in a data track or locate a relation name associated with a given item, (2) store an item or a relation name in the file, and (3) rearrange the elements in a list of link or label blocks to keep them in the proper order.

<u>3.2.1. LOCATE(LINKAD2,RELATION,ITEM2,OFFLIST, MULTFLAG,PTPOS) BIT(1)</u>; The declarations of the parameters are as follows:

```
        DCL 1 LINKAD2,
                2 TKA BIT(15),
                2 ØFST BIT(8),
              1 RELATION,
                2 LBNAME BIT(7),
                2 MUL BIT(1),
              1 ITEM2,
                2 ITK# BIT(15),
                2 IØST BIT(8),
              OFFLIST BIT(1),
              MULTFLAG BIT(1),
              1 PTPØS,
                2 PTK# BIT(15),
                2 POFST BIT(8),
                2 PINDEX BIT(8);
```

This function locates an item or a relation name in a segment of link (or label) blocks and returns '1'B if the item or a relation name is in the segment and '0'B otherwise. A list of link (or label) blocks may be broken up into several segments which reside on different tracks. The elements in the list are arranged in descending order according to the internal values of items or relation names. LOCATE is a basic routine which does the following things depending on the values passed to the parameters.

(1) If LINKAD2 has a non-zero value, it specifies the item which is to be located in a segment of link blocks. The segment can be the first segment of the list which contains the label block (ITEM2) or in a segment in a continuation track. In the former case, MULTFLAG has value '0'B and the head of the segment is identified by PTPØS which specifies the position of the relation name (RELATION) in the list of label blocks representing the item ITEM2. In the latter case, MULTFLAG has value '1'B and the first block of the segment is identified by ITEM2. If the item is found in the specified segment, the structure PTPØS would contain its location. Otherwise, PTPØS contains the location of an item which would have succeeded the item to be located if it were in the list. PTK# and PØFST in structure PTPØS specify the block in which the item is stored and PINDEX specifies its position in the block. If the internal representation of the item to be located is smaller than the last item in the given segment, the flag ØFFLIST is set and PTPØS contains the location of the last item. MULTFLAG is set to '0'B if the segment contains only one item and '1'B otherwise.

(2) If LINKAD2 is zero, this procedure locates the relation name specified by input parameter RELATION in a segment of label blocks whose head block is specified by ITEM2. Upon returning to the calling routine, PTPØS

specifies the location of the relation name if it is found in the segment. Otherwise, PTPØS specifies the location of the next relation name, i.e. the one whose internal representation is smaller than that of the one to be located. If the internal representation of the relation name is smaller than that of the last one in the segment, the flag OFFLIST is set and PTPØS contains the location of the last relation name.

The search method employed in this program is a combination of so-called 'bucket search' and 'binary search'. The program first determines the block in which the item (or the relation name) should be stored by checking the item (or the relation name) against the last element of each block in the segment. Then a binary search is employed to locate the item (or the relation name) in the block.

3.2.2. STLINK(PTPOS,ITEMAD,LABEL,LISTAD,MULFLAG, OFFLIST) The declarations of the parameters are as follows:

```
DCL 1 PTPOS,
        2 PTK#
        2 POFST BIT(8),
        2 PINDEX BIT(8),
    1 ITEMAD,
        2 DUMMY BIT(1),
        2 MEASURE BIT(8),
        2 LINKAD,
          3 TK# BIT(15),
          3 IOFST BIT(8),
      1 LABEL,
        2 NAME BIT(7),
        2 MULT BIT(1),
      1 LISTAD,
        2 LTK# BIT(15),
        2 HEAD,
          3 LOFST BIT(8),
          3 RN BIT(7),
        2 MUL BIT(1),
      (MULFLAG, OFFLIST) BIT(1);
```

This procedure stores an element (an item or a relation name plus an item) at a given location in a list of link (or label) blocks and, when necessary, rearranges the elements in the list of blocks to keep them in the proper order. If the parameter LABEL.NAME is zero, the procedure stores the item represented by ITEMAD at the location specified by PTPOS. Otherwise, the procedure stores (1) a relation name specified by the structure LABEL and (2) a single item specified by the structure ITEMAD at the position specified by the structure PTPOS. Values for the parameter PTPOS will normally have been obtained by calling the LOCATE routine (See 3.2.1).

If OFFLIST has the value '1'B, then the new item is to be stored on the end of the list, the address of whose last element is contained in PTPOS. Otherwise, PTPOS specifies the position in the list where the new element is to be stored. Notice that this position may currently contain some other element which will have to be moved to accommodate the new one. If NAME is zero, then ITEMAD contains a link to be stored at PTPOS, which will refer to a location in a link block. If NAME is non-zero, then LABEL will be taken to be the name of a relation and ITEMAD will refer to an item, or items, that stand in this relation to the item in one of whose label blocks the new entry is to be made. In this case, the 40 bits obtained by concatenating LABEL with ITEMAD is to be stored at the position referred to by PTPOS, which will be in a label block. When necessary, the procedure reorders the elements in the list of link (or label) blocks. This may involve creating a new block and inserting it in the list, extending an old block, moving elements up or down the list to make room for the new element, activating the garbage collector to obtain space for a new block, or may cause an element already in the list of blocks to be moved to another track in order to make room for the new element. This last situation would require updating the track-continuation directory and storing the element removed from the track in another track.

3.2.3. ADDLINK(PTPOS,ITEMAD,LABEL,REMAINDER,OFFLIST)

The declarations of the parameters are as follows:

```
        DCL 1 PTPOS,
                2 PTK# BIT(15),
                2 POFST BIT(8),
                2 PINDEX BIT(8),
              1 ITEMAD,
                2 DUMMY BIT(1),
                2 MEASURE BIT(8),
                2 LINKAD,
                  3 ITK# BIT(15),
                  3 IOFST BIT(8),
              1 LABEL,
                2 NAME BIT(7),
                2 MULT BIT(1),
              1 REMAINDER,
                2 LASTLK BIT(40),
                2 REMAIN BIT(40),
        OFFLIST BIT(1);
```

This procedure is used mainly by the STLINK procedure. Its purpose is to store an element specified by ITEMAD, or LABEL concatenated with ITEMAD, at the position specified by PTPOS in a list of link or label blocks. If the one-bit OFFLIST is set, the element is to be added to the end of the list. If the addition of an element in the list causes another element to be extended to another track, then this procedure returns two elements specified in the structure REMAINDER. REMAIN specifies the element to be stored in another track. LASTLK is the last element in the current list. These two elements must be used to update the track continuation directory.

3.2.4. PRE_BK(BEGIN,END,TYPE,TPOINTER,SIZE) BIT(1)

The declarations of the parameters are as follows:

```
DCL (BEGIN,END) BIN FIXED(15),
    TYPE BIT(1),
    SIZE BIN FIXED(8),
    TPØINTER PØINTER;
```

This function searches a list of link or label blocks, from the block specified by BEGIN to the block specified by END, for an unused space, and moves all the elements following it up by one space. If TYPE = '1'B, then the space is required in a label, otherwise in a link block. The purpose of this procedure is to create space in a block which is at present full for a new item. To do this, an attempt is first made to find some unused space in a block which precedes this one in the list, and on the same track, and then to move the items in intervening positions up so as to fill this space and to leave space available in the desired block. Thus, BEGIN would index the first block of the list that is on the current track and END would index the block in which it was desired to create space. The value of the function is '1'B if the search for unused space is successful, and '0'B otherwise. If the search is successful, then TPOINTER contains the address of the newly created segment of unused space, and SIZE gives the number of items that it will accommodate. This procedure is used mainly by the FORWARD procedure and the garbage-collection routine.

3.2.5. FORWARD(BEGIN,END,ITEMAD,OFFLIST) BIT(1)

The declarations of the parameters are as follows:

```
DCL (BEGIN,END) BIN FIXED(15),
    1 ITEMAD,
      2 DUMMY BIT(1),
      2 MEASURE BIT(8),
      2 LINKAD,
        3 ITKA BIT(15),
        3 IOFST BIT(8),
OFFLIST BIT(1);
```

This function searches a list of link or label blocks, from the block specified by BEGIN to the block specified by END, for an unused space, moves up all the elements following the unused space found in the list, and stores the new element specified by ITEMAD in the space just vacated. If OFFLIST is set, the new element is added at the position following the last element on the list. This function returns '1'B, if it succeeds in finding an unused space in the list, and '0'B otherwise. This function makes use of the PRE_BK function.

3.2.6. POST_BK(BEGIN,END,ITEMAD7) BIT(1)

The declarations of the parameters are as follows:

```
DCL (BEGIN,END)BIN FIXED(15),
      1 ITEMAD7,
        2 DUMMY BIT(1),
        2 MEASURE BIT(8),
        2 LINKAD,
          3 ITK# BIT(15),
          3 IOFST BIT(8);
```

This function searches a list of link or label blocks, from the block specified by BEGIN to the block specified by END, for an unused space and moves down all the elements preceding it by one space, and stores a new element at the space vacated. This function returns '1'B, if an unused space is found and returns '0'B otherwise. Typically, this function is used to obtain space only when an attempt to do so using FORWARD has failed.

3.2.7. UP(MCSTART,MC) The declarations of the parameters are as follows:

```
DCL (MCSTART,MC) BIN FIXED(8);
```

This procedure moves the elements of a block up by one space. MCSTART and MC specify the first and the last elements in the block that are to be moved.

3.2.8. DOWN(BT,TOP,INC,TEMP) The declarations of the parameters are as follows:

```
DCL (BT,TOP,INC) BIN FIXED(8),
    TEMP POINTER;
```

This procedure moves the elements of a block (label or link) down by the number of spaces specified by INC. TOP and BT specify the first and the last elements in the block that are to be moved. TEMP is the pointer of a based variable overlaid on the block.

3.2.9. GETRACK(TRACKNØ) The declaration of the parameter is as follows:

```
DCL TRACKNO BIT(15);
```

This procedure checks if the track specified by TRACKNO is in core. If the track is not already in core storage, it reads the track from disk. This may involve writing a track in core out to disk to make room for the track to be brought in. After the track is located in core, the program assigns its core address to the external pointer variable QTK if a data track is involved, and to the external pointer variable R if a directory track, and, if the track is a new one brought into service for the first time, initializes the structure.

3.2.10. OUTRACT This procedure is used only for debugging purposes. It prints the contents of the track which is currently in core and whose address is the value of the external pointer variable QTK.

<u>3.2.11 STORE(MCOUNT,ITEMAD)</u> The declarations of the parameters are as follows:

```
DCL  MCOUNT BINARY FIXED(8),
     1 ITEMAD,
       2 DUMMY BIT(1),
       2 MEASURE BIT(8),
       2 LINKAD,
         3 L_TRACK BIT(15),
         3 L_BLOCK# BIT(8);
```

The procedure also refers to the following external variables:

```
DCL  QTK POINTER EXTERNAL,
     LB7 BIT(7) EXTERNAL,
     1 TKSTATE(5) EXTERNAL,
       2 T_TRACK BIT(15),
       2 STATUS BIT(1),
     QUEUE(5) BINARY FIXED EXTERNAL;
```

This procedure stores a label or link element in a track. If LB7 has the value zero, then ITEMAD is a link item; otherwise LB7 is the name of a relation and a label item is stored. The position in the track at which the new item is to be stored is given by MCOUNT. QTK contains the address of the track in which the item is to be stored. TKSTATE and QUEUE each contains an entry for each track currently in core; T_TRACK gives the numbers of these tracks (i.e. their addresses on disk) and STATUS contains '1'B or '0'B depending on whether the corresponding track has or has not been modified since it was last replaced on disk. QUEUE contains indices in TKSTATE and is used to order the tracks in core according to how recently they were last referred to. Thus, T_TRACK(QUEUE(1)) is the number of the track most recently referred to, and T_TRACK(QUEUE(5)) is the number of the track that was referred to longest ago. When the STORE procedure is called, QTK is assumed to contain the address of the track most recently referred to, that is, the one corresponding to TKSTATE(QUEUE(1)). The procedure sets STATUS(QUEUE(1)) = '1'B to show that the track has now been modified.

3.2.12. AVL(LASTPLOC,TYPE,FIRSTPLOC) The declarations of the parameters are as follows:

```
DCL (LASTPLOC,FIRSTPLOC)BIT(13),
    TYPE BIT(1);
```

This procedure obtains space from the space available on a data track either for forming a new link or label block or for expanding an old link or label block. TYPE is set to '0'B or '1'B depending on whether the space is required for a link or a label block.

This procedure is called when space is needed for storing a new element in a list of link or label blocks. As noted in the description of STLINK, there are two ways to obtain the space. One is to expand the size of an old block in which to store the new element. The other method is to establish a new block and to link it to the existing blocks in the list. The program has to consider the cost of expanding the size of an old block, which requires rearranging some blocks in the track so that the space consecutive to the old block can be made available. The amount of labor involved in increasing the size of an existing block depends on the number of blocks intervening between it and the available space. Thus, when AVL is called, LASTPLOC contains the address of the last byte in the old block. If the distance between the byte and the first byte of the available space exceeds a predetermined system parameter (currently set to 40 bytes), space for a new block will be obtained. Otherwise, space will be obtained for increasing the size of the old block.

Upon returning to the calling program, TYPE is set to '0' or '1'B according as the space is for forming a new block or for expanding an old block, and FIRSTPLOC is set to point to the space obtained.

3.3. Directory-Track Procedures

The procedures to be described in this section carry out the following operations on directory tracks:

(1) Search the track-continuation information stored on directory tracks to determine the location in a data track at which a relation name or an item is stored.

(2) Establish a new element in a directory track when a list of link or label blocks in a data track is extended to another data track, and

(3) Rearrange the information in directory tracks to keep elements in proper order and to keep all continuation information pertaining to a data track in the same directory track.

3.3.1. SLTRACK(LINKAD,LISTAD,ITEM,PTPOS) BIT(1);

```
DCL 1 LINKAD,
        2 DUMMY BIT(1),
        2 MEASURE BIT(8),
        2 IT,
          3 TK# BIT(15),
          3 ØFST BIT(8),
    1 LISTAD,
        2 LTK# BIT(15),
        2 HEAD,
          3 LØFST BIT(8),
          3 RN BIT(7),
        2 MUL BIT(1),
1 ITEM,
  2 ITK# BIT(15),
  2 IØFST BIT(8),
1 PTPØS,
  2 PTK# BIT(15),
  2 PØFST BIT(8),
  2 PINDEX BIT(8);
```

This function searches a directory track to find the proper segment of a list of link or label blocks (specified in the structure LISTAD9) which contains the link specified by LINKAD or the relation name specified by LISTAD.RN. In the latter case, LINKAD is zero. If the list of blocks does not have a continuation block in another track, this function sets ITEM equal to the first block of the list (a label block) and returns '0'B. Otherwise, it sets ITEM equal to the first block of the selected segment and PTPOS equal to the address of the element in the directory track which specifies the address of the next segment of the list, and returns '1'B.

3.3.2. LOCDRT(LISTAD,TEMPOF,IND,LASTLINK,FALLOFF) BIT(1)

```
DCL 1 LISTAD,
      2 LTK# BIT(15),
      2 HEAD,
        3 LOFST BIT(8),
        3 RN BIT(7),
  2 MUL BIT(1),
TEMPOF BIN FIXED(8),
IND BIN FIXED,
1 LASTLINK,
  2 MEASURE BIT(9),
  2 IT,
    3 TK# BIT(15),
    3 OFST BIT(8),
FALLOFF BIT(1);
```

This function locates a piece of track continuation information in a list of fixed-length blocks in a directory track. The address of this track is assumed to be the current value of the external pointer variable R. The piece of information concerns the track continuation of a list of link or label blocks in a data track. The address of the list is specified by LISTAD, and the last element in the list is specified by LASTLINK. LISTAD and LASTLINK make up the element to be sought. If the element is found in a list of blocks in the directory track, then this function returns '1'B. TEMPOF specifies the block containing the element and IND specifies the position of the element in the block. If the element is not found, '0'B is returned, and TEMPOF and IND are set to point to the first element which has a smaller internal value than the one sought. If the internal value of the element sought is smaller than the last element on the list, FALLOFF is set to '1'B and TEMPOF and IND specify the location of the last element.

3.3.3. EXTENT (LISTAD,ITEMAD,TYPE,LASTLINK)

```
DCL 1 LISTAD,
        2 LTK# BIT(15),
        2 HEAD,
          3 LOFST BIT(8),
          3 RN BIT(7),
        2 MUL BIT(1),
      1 ITEMAD,
        2 TK BIT(15),
        2 OFFSET BIT(8),
      TYPE BIT(1),
      1 LASTLINK,
        2 DUMMY BIT(1),
        2 MEASURE BIT(8),
        2 ITM,
          3 TK# BIT(15),
          3 OFST BIT(8);
```

This procedure establishes a new continuation block or finds an existing continuation block in a data track, and updates the track continuation directory. The continuation block is a link block or a label block depending as TYPE is set to '1'B or '0'B by the calling routine. It is used for storing an element which is pushed out of a list of link or label blocks in a data track. The address of such a list is specified by LISTAD and the last element in the list is given by LASTLINK. The information in LISTAD is used to locate the proper element in a directory track which contains the continuation information on the list. The content of LASTLINK is used to update the continuation information. Upon returning to the calling procedure, ITEMAD contains the location of the block established or found.

If no continuation information related to the list has previously been stored in a directory track, the list specified by LISTAD is now extended to another track for the first time. In this case, a new element is to be established in a directory track. This procedure calls LOCDRT (see 3.3.2) to find the proper place to establish the new element. This generally involves storing a new element in a list of fixed-length blocks in a directory track. Like storing a new element in a list of blocks in a data track, the insertion of a new element in a directory track often requires the rearrangement of the old elements in the list to keep them in the proper order. The program first looks for an unused space in the block located by the procedure LOCDRT. If a space is found, the program rearranges the elements in the block and inserts the new element. If this fails, the program gets a new block from the available space in the directory track and links the new block to the other blocks in the list. The new element will then be properly stored.

In the event that there is no space in the directory track available for establishing a new block, the program will make room by moving another list of blocks from the present directory track to another directory track instead of extending the current list to another track. This is because we want to keep all the track continuation information associated with a data track in the same directory track. Generally, the shortest list will be moved out and the blocks returned to the available space. In this way, the new block can be established in the present directory track.

3.3.4. SUCBK (BEGIN,END) BIT(1)

```
DCL (BEGIN,END) BIN FIXED(15),
    (TEMPØF,IND) BIN FIXED EXT(15);
```

This function searches a list of fixed-length blocks in a directory track, from the block specified by BEGIN to the block specified by END, for an unused space and moves all the elements preceding it down by one space. An empty space will thus be created in the first block of the list. Upon returning to the calling routine, TEMPØF specifies the block containing the empty space and IND specifies the position of the empty space in that block. If the program fails to find any unused space in the list, it returns '0'B. Otherwise it returns '1'B.

3.3.5. PREVBK(BEGIN,END)BIT(1)

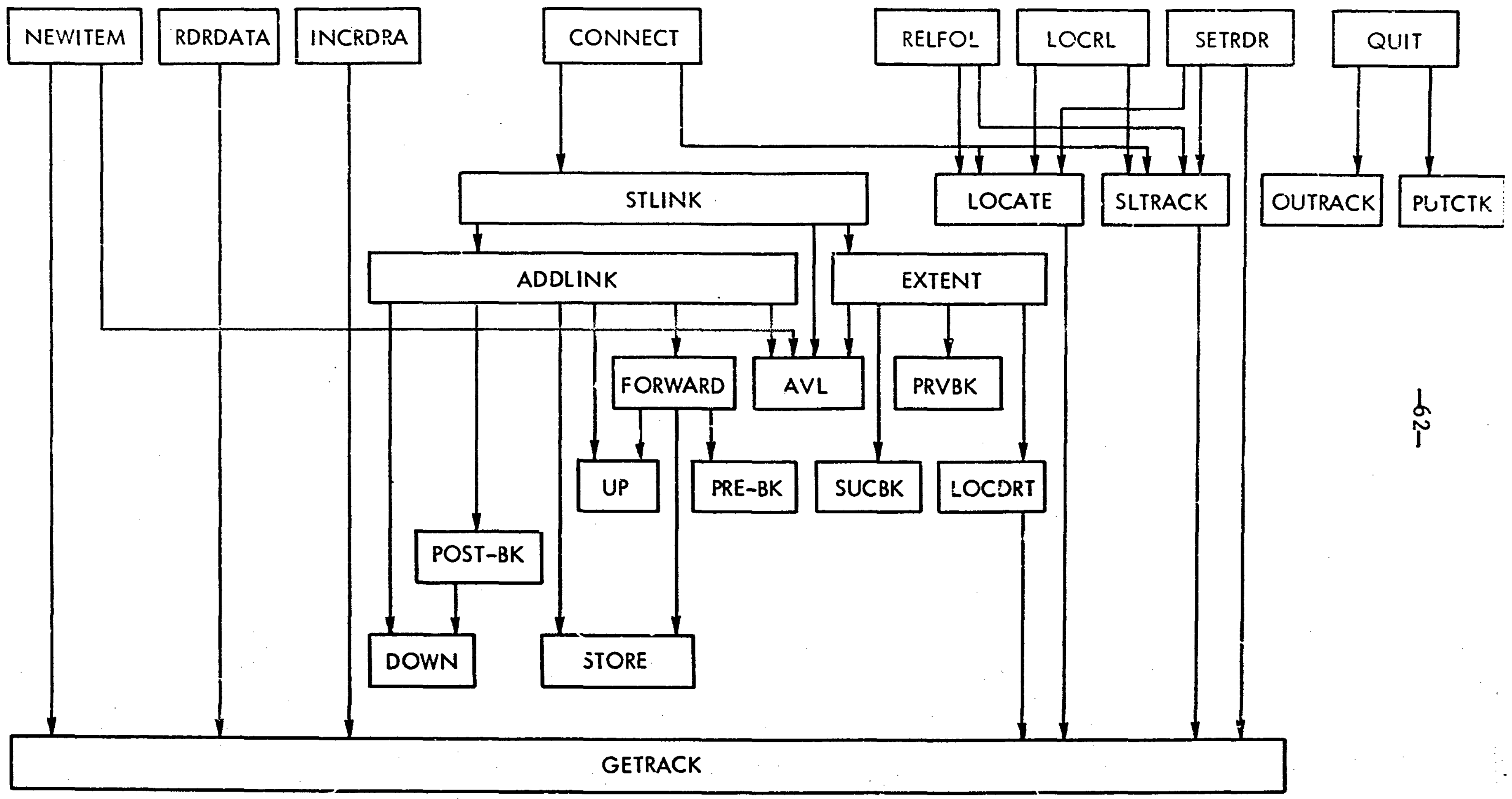
```
DCL (BEGIN,END) BIN FIXED(15),
    (IND,TEMPOF) BIN FIXED EXT(15);
```

This function searches a list of fixed-length blocks in a directory track from the block specified by BEGIN to the block specified by END, for an unused space and moves all the elements following the unused space in the specified portion of the list up by one space. Thus, an empty space will be created at the end of the list. Upon returning to the calling routine, TEMPOF specifies the position of the block containing the empty space, and the position in the block is of the space identified by IND. If the program fails to find any unused space in the list, it returns '0'B. Otherwise it returns '1'B.

3.3.6. PUTCTK This procedure prints a directory track whose address is the current value of the external pointer variable R. It prints the content of all the fixed-length blocks in the directory track except the empty ones.

4. PROGRAM STRUCTURE

The following chart specifies the relations among the procedures described in this report showing which procedures make use of which others. An arrow in the chart indicates that a procedure calls another procedure to which the arrow points.


NEWITEM
RDRDATA
INCRDRA
CONNECT
RELFOL
LOCRL
SETRDR
QUIT
STLINK
LOCATE
SLTRACK
OUTRACK
PUTCTK
ADDLINK
EXTENT
FORWARD
AVL
PRVBK
UP
PRE-BK
SUCBK
LOCDRT
POST-BK
DOWN
STORE
GETRACK