Technical Report CS74001-R

# A FILE DEFINITION FACILITY FOR FILE STRUCTURES

Billy G. Claybrook

May 1974

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

## ABSTRACT

This paper describes a file definition facility (FDF) for defining files as graph structures. The structure of the file is explicitly declardd in the file definition. Primitive functions(from graph theory), operators, and the format of the definition statements are given. The combination of functions and operators appear as directives to the programming system for structuring files.

Several simple examples are given to illustrate the use of the FDF. The data organization for the implementation of this facility is described in detail. Problems of considerable importance that are treated are: (1) garbage collection, (2) template construction, and (3) runtime address calculation. The external definitions are represented internally by descriptors. The format of the descriptors is given and a discussion of the items in the descriptors is presented.

## 1.0 INTRODUCTION

Many applications in Information Storage and Retrieval, data base design, and other file structuring related areas require the use of files organized in arbitrary ways. Any change in file structure design may require a sufficiently large amount of recoding so that it becomes inconvenient to make the change. Current file definition facilities in most programming languages give little, if any, description of the underlying structure of the file itself. Languages such as APL[5], FOL[2], and IDS[1] that allow file structures more general than trees do not have the capability to allow the user to describe the structure of his files explicitlyin the file declaration. We feel that it is both descriptive and instructive for a programming language to have the capability just described. It is instructive inthe sense that it can "instruct" the programming system to perform certain operations automatically, both on the file and on the records in the file.

The desire to have a file definition facility (FDF) that allows the user to explicitly (and arbitrarily) describe his file structures necessitates the development of some type of notation for doing this. The notational scheme should be of such a nature that it can be efficiently implemented in a programming system. Besides using the notation in an instructive manner, e.g. that described above, we feel that such a notation is important for communicating file structure descriptions between programs or between humans. The FDF initiated in this paper can be extended to serve as a communicating element and also as a descriptive and instructive element interpreted by a programming system to structure files. Therefore, the major incentive for this research is to develop a "semiformal" means for defining graph structured files with variably formatted record elements and then be able to efficiently implement the FDF in a programming system.

The FDF described in this paper is based on a graph structure view of files. Codd [4] describes the inadequacies of tree structured files or more general network models of data, e.g. graphs. However, he recognizes that many data bases are organized along these lines. The development of a FDF for describing general file (or data) structures brings with it a few potential problems, e.g. garbage collection, list traversal, reentrant structures, etc. Also associated with the desirability to have generalized file structures is the need for record elements to have descriptions that can vary dynamically in size and format and be referenced individually. So what we are proposing is a FDF that necessitates the use of garbage collection, requires templates to describe dynamic records, and needs a powerful means for describing records with arbitrary types of components.

This type of system might appear, to the reader, to be complex and inefficient (with respect to space and time) and thus inappropriate for any practical use. This system is predicated on the fact that virtual memory systems are now available for use and by the fact that computer systems, especially microprogrammable ones, are replacing traditional software functions by microcodes. But we do not depend entirely on virtual memory and microprogramming alone to make this facility feasible. Many inefficiency problems can be overcome by the development of efficient programming techniques. In Section 4.0 we describe a scheme for efficient garbage collection that is currently implemented in LPL [3] . The other potential problems such as list traversal, reentrant structures, and templates to describe records are also treated in Section 4.0.

The FDF makes use of several primitive functions from graph theory for the description of files. Operators are defined for use in record and file definitions. The primitives, the operators, and the format of the definition statements are given in Section 2.0. In Section 3.0 we give examples of structures defined using the FDF. Finally, Section 4.0 describes the data organization suggested for implementing this facility.

## 2.0 NOTATION AND SYNTAX

This section defines the primitive functions and the operators used in the FDF. The author by no means deems this is a complete set of functions and operators for defining all graph structures; thus, he expects more functions and operators to be defined during attempts to extend the FDF. Throughout the rest of this paper x is a file, y is a record, and z is a structure.

### 2.1 Primitive Functions

Each primitive function is given below and the argument type(s) that it requires are given. When x and y are used as arguments in a function, we are not referring to the file or record descriptors (see Section 4.0) associated with each, but instead to a particular allocation of each.

1. cycle (pointer$_1$, pointer$_2$, ..., pointer$_n$), or

   cycle (pointer$_1$(key$_1$), pointer$_2$(key$_2$), ..., pointer$_n$(key$_n$))

   pointer$_i$ is the name of a pointer field defined in a record definition. pointer$_i$(key$_i$) is the name of a pointer field defined in a record definition and associated with a particular key. key$_i$ can be a KEY variable [3] or alternatively key$_i$ can be an index into an array of pointer variables. Either way, key$_i$ identifies a particular pointer.

   The cycle function indicates that a cycle (or loop) exists in a file. A cycle is defined as the loop created by traversing the structure linked by a particular pointer given in the cycle argument list. If the cycle function has three arguments, then three individual cycles exist. These cycles may or may not have records in common.

2. head(x)*

   The head function refers to the head record of file x.

3. tail(x)*

   The tail function refers to the tail record of file x.

4. son(y)

   son implies an immediate descendent of y. The son function can imply a set of records as well as an individual record. The number of records in the son-set is less than or equal to outdeg(y).

5. successor(y)

   successor has the same interpretation as son. Both functions are given to allow a choice in terminology on the part of the user, e.g. the author prefers son when describing tree structures and successor when describing list structures.

6. father(y)

   father implies an immediate ancestor. The father-set has a maximum of indeg(y) records in it.

7. predecessor(y)

   predecessor has the same interpretation as father. father and predecessor have the same relationship as son and successor.

8. ancestor(y)

   ancestor refers to those records that precede y in a file structure. In reentrant graph structures it is sometimes difficult to determine the ancestor-set of y.

9. descendent(y)

   descendent refers to those records that follow y. As with the ancestor interpretation, it is not always clear exactly what records are in the descendent-set of y.

*In some file structures it is difficult to discern the head and tail of a file.

10. filial(y), filial(y, arcname)

    filial(y) implies the set of terminal records of all arcs (or links) originating in y. filial(y, arcname) is the set of terminal records of all arcs that originate in y and bear the label arcname. For our purposes arcname is a key associated with a particular pointer, or it is an index into a pointer array.

11. indeg(y)

    indeg is the number of branches (or arcs) entering record y (indeg is related to the reference count). indeg can have more than one actual parameter.

12. outdeg(y)

    outdeg is the number of branches leaving record y. outdeg(y) is equal to the number of pointer components in a record. outdeg can have more than one actual parameter.

## 2.2 Operators

The notation x.y used below indicates qualification, i.e. y is a record in file x.

1. pointer $\longrightarrow$ record

   $\longrightarrow$ is the points to operator

   Example: head(x).p(1) $\longrightarrow$ tail(x)

2. $\text{record}_1 \longleftarrow \text{record}_2$

   $\longleftarrow$ is the replaced by operator

   Example: tail(x) $\longleftarrow$ x.y

3. = is the assignment operator.

   = can assign a specific value to a field in a record, or it can specify a value to be entered in the descriptor associated with a record or file. If the assignment specifies a value for a field in a record, the assignment is done automatically upon allocation of the record. For example: sysid=2, sysrc=3 (the sysid and sysrc fields common to every record are discussed in Section 4.0). In the case of the indeg and outdeg functions, these values are specified in the file or record descriptors only.

4. $\equiv$ is the is defined as operator.

$\equiv$ equivalences two items, e.g. head(x) $\equiv$ y.

5. $\wedge, \vee, \neg$

These operators have the same meaning as the Boolean and, or, and not operators. An example from Section 3.0 illustrates their use:

y.p(VAR) $\rightarrow$(ancestor(y) $\vee$ descendent(y))

## 2.3 Syntax for Definition and Allocation Statements

The syntax of the statements given in this section is not in BNF notation, but instead it is given via simple examples. At this point in the development of the FDF, we do not feel that it is important to provide the syntax in BNF notation. The syntax resembles, to some extent, that used by ALGOL 68 [8] for defining structures. The most basic definition described in this paper is the structure definition:

structure z = (1 i integer, 1 a real, 1 value, 2 b real, 2 p(5) pointer)

The structure definition is equivalent to the PL/1 structure with two exceptions: (1) an FDF structure can have another structure(s) as a component (we do not allow a structure to have itself as a component either directly or indirectly), and (2) an FDF structure can have components with variable dimensions defined at runtime.

Record definitions exist as follows:

record y = (r pointer, structure z, sysid=2)

This definition of record y says that it has structure z and pointer r as components. Also, the system identifier field (sysid) in record y is initialized to 2 automatically for each allocation of y. The record definition can also include primitive function directives.

The *file* definition specifies all records that will be elements in a file. For example:

*file* x = (*record* y, *indeg*(y)=2, *outdeg*(y)=2, *cycle*(y.p(1), y.p(2)))

*file* x' = (*record* $y_1$, $y_2$, $y_3$, *indeg*($y_1$, $y_2$) = 2, *indeg*($y_3$) = 1, *outdeg*($y_1$, $y_2$) = 2, *outdeg*($y_3$) = 1, *cycle*($y_2$.p(1), $y_2$.p(2)))

The last statement type that we discuss is the *alloc* statement. It can take on various forms, one of which is

*alloc* y, ptr;

Here we are allocating *record* y with ptr pointing to it (ptr must be a *pointer* variable). Descriptive information can also be included in an allocation statement. The descriptive information can indicate where the record is to be placed in the file; or if some directives were not placed in the *record* or *file* definitions, then they can be placed in parentheses immediately following the record pointer (or type indicator), e.g.

*alloc* y, ptr(*tail*(x) $\longleftarrow$ x.y);

A type indicator (must be an integer variable or constant) can be associated with a particular instance of a record (the use of the type indicator is described in Section 4.0). The form of the allocate statement with the type indicator is:

*alloc* y, ptr, itype;

Section 4.0 gives the internal representation of *structure*, *record* and *file* definitions in the form of descriptors. The next section illustrates the use of the FDF by describing some simple structures.

## 3.0 EXAMPLES

The use of the FDF is illustrated in this section by a series of examples describing various data structures:

1. Sequential file, singly-linked.

   file x = (record y, outdeg(y)=1, indeg(y)=1, tail(x) $\leftarrow$ x.y);

2. Sequential file, singly-linked, circular.

   file x = (record y, outdeg(y)=1, indeg(y)=1, tail(x) $\leftarrow$ x.y,
   tail(x).p $\rightarrow$ head(x));

   record y = (p pointer, value integer);

3. Binary tree file.

   file x = (record y, outdeg(y)=2, indeg(y)=1, y.p(1) $\rightarrow$ son(y),
   y.p(2) $\rightarrow$ son(y))

   record y = (p(2) pointer, value integer);

4. Sequential file, doubly-linked (SLIP-like fashion [9] )

   file x = (record y, outdeg(y)=2, indeg(y)=2, y.p(1) $\rightarrow$ predecessor(y),
   y.p(2) $\rightarrow$ successor(y), head(x).p(1) $\rightarrow$ tail(x), tail(x).p(2)
   $\rightarrow$ head(x))

   record y = (p(2) pointer, value integer)

Another way to define a SLIP file structure is to indicate that the head is a separate node, say $y_1$. Then

file x = (record $y_1$, $y_2$, outdeg ($y_1$, $y_2$)=2, indeg($y_1$, $y_2$)=2,
$y_2$.p(1) $\rightarrow$ predecessor(y), $y_2$.p(2) $\rightarrow$ successor(y),
head(x) $\equiv$ $y_1$, $y_1$.p(1) $\rightarrow$ tail(x), tail(x).p(2) $\rightarrow$ $y_1$)

record $y_1$ = (p(2) pointer)

record $y_2$ = (p(2) pointer, value integer)

When a particular record is specified as a head record, it must be allocated just like any other record, e.g. alloc $y_1$, ptr; .

5. Graph file.

file x = (record y, outdeg(y) = VAR, indeg(y) = VAR, y.p(VAR) $\longrightarrow$ (ancestor(y) $\vee$ descendent(y)));

record y = (p(M) pointer, value integer);

The definition of the graph structure above introduces a new symbol, VAR. The reserved symbol VAR is used to indicate that a variable number of some item exists. VAR never has any value associated with it. The value of M in the record definition must be specified by the user at execution time.

6. Tree file.

file x = (record y, outdeg(y)=VAR, indeg(y)=1, y.p(VAR) $\longrightarrow$ son(y));

record y = (p(M) pointer, value integer);

The above structures are not the only ones describable by the FDF; they were chosen because most readers are familiar with each structure. We have purposely kept the record definitions simple. At this point we could explain how the directives in the definitions are actually used to help structure files, but any explanation requires familiarization with the descriptors for definitions. So we defer this discussion until Section 4.0.

## 4.0 DATA ORGANIZATION FOR AN IMPLEMENTATION

Since the file structures are graph structures, the internal representation of a file is a list of records. Each record has the following fields in common:

(1) a type field (TYPE) that contains a pointer to the template describing this type of record,

(2) a copy bit (COPY) for use in copying files, especially reentrant files,

(3) a system reference count field (SYSRC),

(4) a system identifier field (SYSID), and

(5) two link fields, MLP and MRP, that link all records allocated by the user into a doubly-linked "super list" for use by the garbage collector.

Some of the fields common to all records are self-explanatory; therefore, we discuss only the SYSID and SYSRC fields and the use of the super list. Complete details and descriptions of the common fields and record manipulation statements are available in [2] and [3].

One problem associated with processors that allow generalized data structures to be defined is gargage collection. Garbage collection is a problem for languages like ALGOL 68 and LPL (and for the FDF) because file traversal for file structures with a variable number of link fields per record is a problem. Fenichel [6] discusses this list traversal problem in detail. We try to minimize the garbage collection difficulties in the FDF by using the system reference count (SYSRC) and the super list (this scheme also appears in LPL). The number of pointer references to a record is maintained by the FDF system in the SYSRC field of each record. As long as the SYSRC is greater than zero, the corresponding record is active and is referenced by one or more records. During garbage collection the super list is traversed and the storage occupied by inactive records is released. Reference count garbage collection schemes are, in general, not capable of freeing records in recursive files because the refernce counts of the records will be nonzero (since they refer to themselves). We can handle this problem by requiring the user to erase the file before he destroys the pointer to it. The erase routine uses the COPY bit and looks at the SYSRC in each record to determine whether a record in a recursive file should be placed in inactive status. Important things to note about this type of garbage collection are:

(1) that the super list alleviates the system from maintaining a list of pointers to all accessible files created by the user (actually this is not aproblem in the FDF), and (2) that the tracing of accessible files during garbage collection is avoided and the marking phase is avoided.

The SYSID field is used to specify the pointer structure in a record. This field requires three bits to specify whether the pointer structure in the record is singly-linked (SYSID=0), doubly-linked (SYSID=1), left-right-linked (SYSID=2), multi-linked (SYSID=3), or key-linked (SYSID=4). Knowledge of the pointer structure by insert (and delete) -type statements is necessary if they are to automatically handle the manipulation of pointers. Key-linked pointer structures associate a key with each pointer. Then the user can specify, by presenting a key, which pointers to access. Multi-linked records present a special problem because there is no automatic way, in general, to determine which pointers to modify during insertion and deletion. For instance, if we want to insert <u>record</u> $y_2$ after <u>record</u> $y_1$ and <u>outdeg</u>($y_1$)=5, then we have to specify which of the five pointers should point to <u>record</u> $y_2$. We could assume that all five would be modified to point to $y_2$ but that is usually not what the user wants to do, especially when working with multi-list files [7] .

## 4.1 <u>Descriptors and Templates</u>

We use the examples of <u>structure</u>, <u>record</u>, and <u>file</u> definitions given in Section 2.3 for describing the internal representation of descriptors. The descriptor for <u>record</u> y appears in Fig. 1. The name of the record references a descriptor describing the definition (the length of the descriptor is determined at compile time). The record descriptor contains the descriptive information included in the record definition.

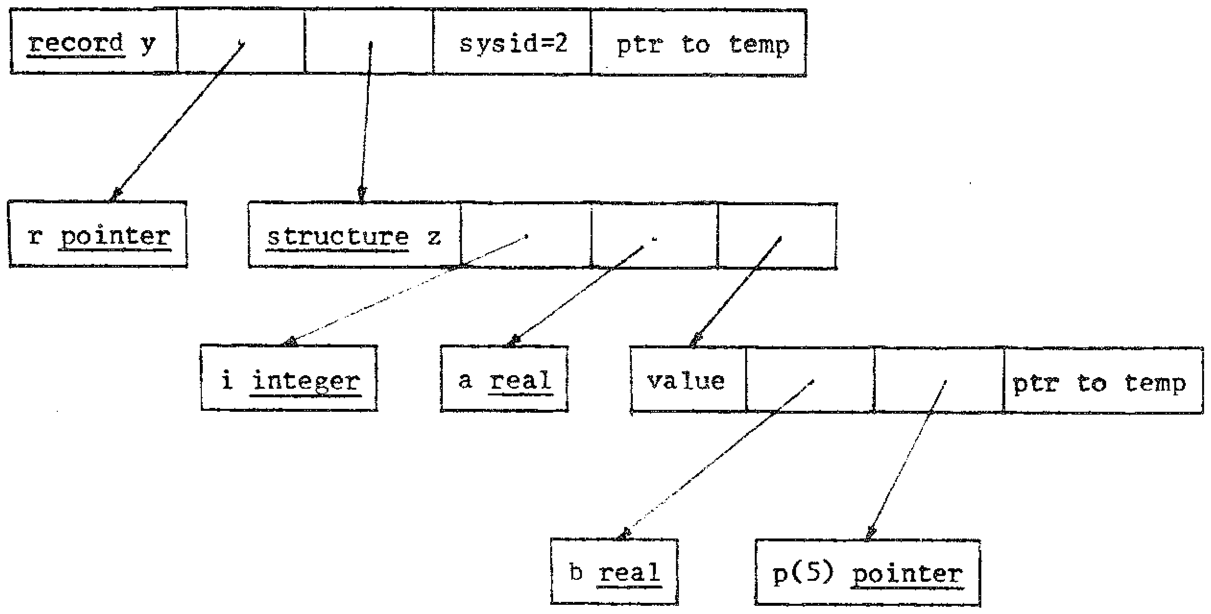

Fig. 1. Descriptor for <u>record</u> y

Each structure has its own descriptor. The third field of the record descriptor is a pointer to the descriptor for z (the structure descriptor is shared by all records containg z). The names of items are placed in the descriptors in Fig. 1 for illustrative purposes.

If a record y' is defined as below with variably dimensioned components, defined at runtime, then a set of templates is required for describing the characteristics of different instances of y' ( such a record is hereafter called a <u>dynamic record</u>). We require the templates not only for traversal and copying purposes but also for freeing the storage occupied by inactive instances of y'. Each record descriptor for dynamic records has an entry containing a pointer to the list of templates for the record (non-dynamic records and structures also have a pointer to a template; however, they have a single template and not a list of templates).

Record y' defined below is an example of a dynamic record. The record and structure descriptors are given in Fig. 2.

record y' = (r(N) pointer, structure z', sysid=2);

structure z' = (1 i(M) integer, 1 a(L) real, 1 value, 2 b(20) real,
2 p(K) pointer);

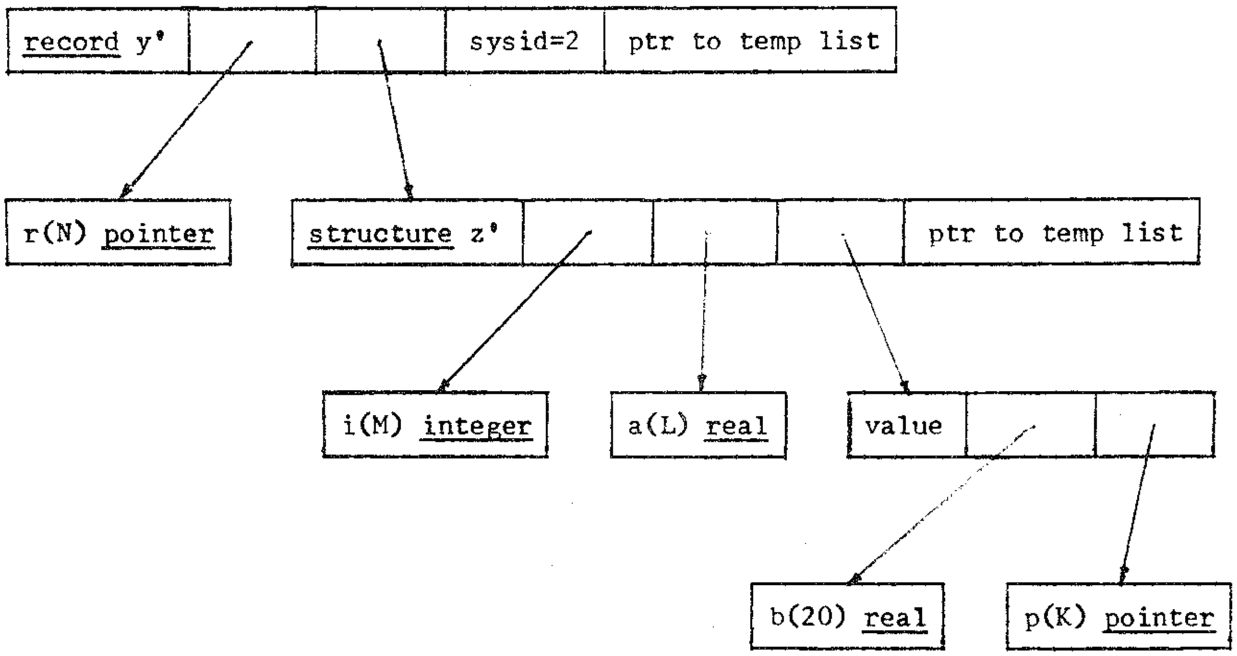

Fig. 2. Record and structure descriptors for y' and z'

When a dynamic record is allocated, efficiency can be improved by specifying the type of the record (don't confuse this with the TYPE field), i.e. alloc y', ptr, itype, where itype is an index into the record template list for y'. If itype is not specified, then the system searches for a template matching the characteristics of record y'. We require this because the TYPE field in each record points to its template. Each record must know where its template is because it has to be accessed when files are traversed, copied, etc. If a template is not located, then one is allocated and filled (the size and configuration of the template can be determined at compile time). The itype parameter appears to be convenient because many users will have a specified number of different types of records that they are using, and a natural way to identify the various types is to assign an integer quantity as an identifier. Note that several types of records can be associated with a single dynamic record definition.

The template(s) for record y' defined above have the format given in Fig. 3. The first word of each template contains the number of words in the template not counting the first and the block size for the particular instance of the record. After word 1 in the template, the left half of each word contains a code indicating the type of entry in the corresponding right half. Word 2 contains a pointer to the next template on the list, if one exists. Entries appear in the templates in the order they appear in the definition statements; i.e. if three structures appear in a record definition, they will appear in the template for that record in the same order. If a record is dynamic but contains a structure definition that is not dynamic, there is still an entry in the record template for it.

| | |
|---|---|
| # of entries | block size |
| TEMPTR | temp link |
| POINTER | value of N |
| STRUCT DESCRIPT | ptr to descript for z' |
| STRUCT TEMP | ptr to struct temp |

Fig. 3. Record template for y'

Fig. 4 gives the template for structure z'. Basically, it has the same format as the record template in Fig. 3. The template in Fig. 4 has five entries and this particular instance of z' requires a block of 46 words. The last four entries indicate that M=15, L=4 and K=7 (M, L and K are the variable dimensions in the <u>structure</u> z'). The left half of each entry indicates the type of entry and the right half contains the quantity. Two things are important about the template in Fig. 4. First, the components in z' are placed in the template in depth-first, left-right order. This order is important because it facilitates address calculations. Secondly, the static components of a structure are also entered into the template.

| | |
|---|---|
| 5 | 46 |
| TEMPTR | 0 |
| INTEGER | 15 |
| REAL | 4 |
| REAL | 20 |
| POINTER | 7 |

*Fig.4. Structure template for z'

Having the block size of a record or structure defined in the template simplifies dynamic storage allocation considerably. The sequence of steps in the execution of <u>alloc</u> y', ptr, itype; follows:

(1) the descriptor of y' is used to locate the template list (indirect addressing set up at compile time can be used to access the template list),

*The zero in word 2 of the template indicates the end of template list.

(2) itype is used to locate the correct template in the list,

(3) the record block size is extracted from the first word of the template, and

(4) a block of this size (given in step (3)) is allocated with ptr pointing to the first word of the block.

Another problem that occurs with the use of dynamic structures and records is address calculation. If the structures or records are not dynamic, then offsets for components can be determined at compile time. The offset is relative to the base address of the storage block allocated at runtime. But, if dynamic structures and records are involved, then the template must be interrogated at runtime to determine the correct offsets. Even with dynamic elements, the machinery for address calculation can be set up at compile time. The following example from structure z' is used to illustrate this. Suppose a reference to component b(j) of structure z' occurs, e.g. z'.value.b(j). We write T(i) to reference the right half of the ith entry in template T (template T is the one in Fig. 4). In order to determine the offset of b(j), the expression T(3) + T(4) + j is generated at compile time. The absolute address is determined by adding the base address of the block allocated for z' and the value of the expression. If structure z' is allocated as part of record y', then the value N must be added to T(3) + T(4) + j to calculate the correct offset since a contiguous block of words is allocated for each record. Note that the descriptor for structure z' plays an important role in determining the offset expression T(3) + T(4) + j. The reader should now appreciate the reason for ordering the template entries as we did.

Instead of illustrating the descriptor for file x since its format is very similar to that of record y, we describe how the directives, e.g.

primitive functions, in the definitions are actually used. This discussion will provide the reader with additional details on descriptor formats and the entries in the descriptors. We use the examples in Section 3.0 for our discussion here.

In the second definition of file x in example 4, we have the directive head(x) $\equiv y_1$. When the descriptor for file x is constructed, an entry for the pointer to the head of file x is included (along with an entry that $y_1$ is the head of x). The pointer entry will be filled when record $y_1$ is allocated. A second allocation of $y_1$ causes this pointer entry to be changed; thus, causing a new instance of the file to be created and the previous instance to be lost unless it is assigned to another file. If the first instance of the file is no longer needed, it should be erased. Every file descriptor has an entry for a pointer to the file it describes; however, if there is no explicit definition of the head of a file then the head is the first record allocated for the file. An implementation of the FDF should include file assignment statements of the form file x = file y.

The directive y.p(1) $\rightarrow$ tail(x) (example 4) causes an entry for a pointer to the tail of file x to be placed in the descriptor. This type directive is important when the tail of a file is frequently changing, thus causing one or more pointer fields to be updated to the new tail each time. The two directives head(x).p(1) $\rightarrow$ tail(x) and tail(x).p(2) $\rightarrow$ head(x) necessitate the use of both the head and tail entries in the descriptor during the creation of such structures (SLIP structures). We mention again that not only do the directives provide descriptive information to the user and the system, but they also cause certain operations to be performed automatically during file creation.

## 5.0 SUMMARY

The FDF is an attempt at providing an explicit definition of the structure of files. The FDF provides the user with a considerable amount of flexibility for defining fileand record structures. The implementation of such a facility is not without its problems (discussed in Section 4.0). However, we feel that an implementation of the FDF (along the lines discussed in Section 4.0) can be developed so that it will be of practical use. Some of the implementation ideas already exist in LPL.

It is the author's intention to incorporate the FDF described in this paper into a systems programming language currently under design and implementation.

## REFERENCES

1. Bachman, C.W. and Williams, S.B. "The Integrated Data Store--A General Purpose Programming System for Random Access Memories", AFIPS 1964 FJCC, Vol. 26, Spartan Books, N.Y., pp. 411-422.

2. Claybrook, Billy G. "FOL: A Language for Implementing File Organizations for Information Storage and Retrieval Systems", Presented at the 1973 SIGPLAN-SIGIR Interface Meeting, To appear in SIGPLAN NOTICES.

3. Claybrook, Billy G. "LPL: A Generalized List Processing Language", To appear in the 1974 Proceedings of the NCC&E.

4. Codd, E.F. "A Relation Model of Data for Large Shared Data Banks", CACM, Vol. 13, June 1970, pp. 377-387.

5. Dodd, George D. "APL--A Language for Associative Data Handling in PL/1", AFIPS 1966 FJCC, Vol. 29, Spartan Books, N.Y., pp. 677-684.

6. Fenichel, Robert R. "List Tracing in Systems Allowing Multiple Cell-Types", CACM, Vol. 14, August, 1971, pp. 522-526.

7. Lefkovitz, David. File Structures for On-Line Systems, Spartan Books, New York, 1969.

8. Peck, J. E. L. (ed.). ALGOL 68 Implementation, North-Holland Publishing Company, Amsterdam, 1971.

9. Weizenbaum, J. "Symmetric List Processor", CACM, Vol. 6, September, 1963, pp. 524-544.