A Data Structure Manipulation System

by

Michael J. Rogers

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba

A DATA STRUCTURE MANIPULATION SYSTEM

BY

MICHAEL J. ROGERS

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

ABSTRACT

In the teaching of a data structures course, the instructor typically demonstrates algorithms by manipulating an abstract representation of the data structure on a a blackboard; a procedure that is often error prone. Also, if the instructor wishes to have students write programs to manipulate a data structure for an assignment, the students' programs first have to construct the data structure (typically unsuccessfully) before they can attack the problem.

This thesis gives a general method for storing data structures on secondary storage. Programs which allow the data structure to be stored and retrieved from secondary storage by executing programs are discussed and implemented on an Amdahl 5850 under MVS for the Pascal/VS compiler. Also, a facility for displaying an abstract representation of a data structure stored in secondary storage is discussed.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, M. S. Doyle, for his support and constructive criticism throughout the preparation of this thesis. I would also like to thank H. Ferch and M. Yunik for the time they spent reading this thesis.

Thanks also must go to E. Reimer and R. Knispel of the University of Manitoba Computer Centre for the invaluable assistance that they provided with NAM and Pascal/VS. There were also many people who were patient enough to listen to my ideas and provided helpful criticism. Without their input this thesis would have taken much longer to complete.

Finally, I would like to thank Kathy Stewart for her support and encouragement throughout the preparation of this thesis.

This work was supported in part by an NSERC postgraduate scholarship.

CONTENTS

Chapter I

INTRODUCTION

In teaching a data structures course, the lecturer typically introduces the concept of linked lists by drawing abstract pictures of them and demonstrates the operations that are to be performed on them in abstract terms. However, the lecturer often runs into problems when he starts to manipulate even a simple data structure such as a linked list, as he has to change parts of the picture, often repeatedly. If the lecturer is working on the blackboard, this will often result in his disappearing into a flurry of chalk dust! Also, as the lecturer cannot stand back and get a clear picture of the entire data structure, there is a good possibility that one of his pointers will end up pointing at the wrong node.

Another problem that occurs in teaching data structures courses is that once the students are programming algorithms for the manipulation of data structures, they have to first create the data structure and then perform the required operations on it. Few programming languages have statements in them that allow the user to store a data structure such as a binary tree in a file and then load it back into the program at some later time. If such a facility did exist,

the students could load a pre-created data structure and concentrate on the manipulation of that structure knowing that it was created correctly.

This thesis was motivated by these two problems. The thesis describes the design and implementation of a system which allows a user to store and retrieve data structures from a special type of file. The data structures that are stored in the file could either be loaded into a student's program or displayed graphically for the entire class to see. Chapter two of this thesis gives a brief overview of the various parts of the system, chapter three gives a detailed design, chapter four provides some details of the implementation Chapter five discusses work that is currently in development with this system. Included as an appendix are details on the user level interface to the access method used in this system, NAM (Network Access Method).

Chapter II

SYSTEM OVERVIEW

As mentioned in the introduction, one of the goals of the system is to have the ability to display the data structures of a program as they are being manipulated during the execution of that program. It soon became clear that there are two parts to this problem:

1. The extraction of the data structure from the program
2. The display of the data structure in some abstract form.

Therefore, the system was broken into two parts, the "Extractor" to obtain the data structure from the program and the "Displayer" to present the data structure to the user in a graphical form. These two parts of the system communicate via a file known as the External Data Structure File (abbreviated to EDSF).

The EDSF contains the data structures in a form external to the program that is manipulating them, rather than just the contents of memory that is being used to store the data. The EDSF also contains information as to the nature of the data structures that are stored in it. The use of the EDSF allows the displayer to run independently of the extractor

as all information required to display of the data structure is stored in the EDSF.

If the user wishes to have the data structure placed in memory for manipulation by his program, he would use another part of the system known as the data structure "Fetcher". By combining the fetcher and the extractor, this system allows the user to read and write data structures, a facility that many programming languages do not support. There were two main reasons for choosing to use an intermediate file to pass data structure information around the system:

1. The EDSF allows the various parts of the system to run independently. This means that a particular data structure can be displayed using the displayer for the class to see and then fetched into their programs as they learn how to manipulate it. Also, if the program from which the data structure is being extracted takes a long time to modify the data structure, the displayer would not have to wait for the change to occur before displaying the next state of the data structure. However, with this approach, there should be no problem with running the extracter and the displayer concurrently if desired. To do this, a program would invoke the extractor to extract a data structure from the program, place it in the EDSF, and then invoke the displayer which would then display that data structure.

2. The EDSF permits the user to examine the state of the data structure both forwards and backwards in time. Thus, if the user missed an event that occured in a previous state, he could back up in the EDSF to look at it again.

Thus, the complete system would consist of 4 parts:

1. The External Data Structure File which is at the center of the system.
2. The Extractor for taking data structures from a program and placing them in the EDSF.
3. The Displayer for displaying a data structure in the EDSF on a graphical output device.
4. The Fetcher for copying a data structure from the EDSF in memory for a program to use.

The initial development of the system did not include the displayer. However, the design of the displayer is covered in the chapter on future developments to the system.

Chapter III

DESIGN

3.1 THE EXTERNAL DATA STRUCTURE FILE

The EDSF stores a data structure in a form external to the program which created it. Thus, in order to reconstruct the data structure for display purposes, the EDSF has to contain not only the data in the data structure but also a description of the nature of the data structure itself. While the description of the data structure need only be given once, the data that is in it will change as the program manipulates the structure.[1] Thus, to save storing the information as to the nature of the data structure more than once, the EDSF is broken into two parts: the definition part and the data part.

As the EDSF consists of two distinct parts a decision had to be made as to how to store those parts. The EDSF could have been stored in an IBM partitioned data set (PDS) but this was rejected for two reasons:

1. A PDS only allows a one level index. If a user wanted to build an EDSF library, it would either require several PDS's or a naming scheme for the members that

[1] In this thesis, the term instance will be used to refer to a state of a data structure at one point in time.

would relate the two parts of the EDSF to each other. This could be done by allowing the user to specify a seven character name for the EDSF and then appending either a 'D' or an 'I' to identify the data and index parts respectively.

2. Automatic garbage collection is not done on an IBM PDS. If a member changes in any way, a new copy of that member is made, and the space that the old copy occupied is not recovered automatically. There is a utility program to recover the space in a PDS but if the system crashes while this program is running there is a good chance that the PDS will be damaged badly.

After considering writing my own access method (and rejecting this idea due to the amount of work that would be involved in this), I decided to use the Network Access Method (NAM) [FERCH82] which was developed at the University of Manitoba for use by MANTES (Manitoba Text Editing System). Details of the functions provided by NAM are discussed in appendix A. NAM was picked because it allows the creation of a hierarchical file system within one dataset. Thus, the two parts of the EDSF can be stored as two files under one directory. The names of these files would normally be "INDEX" and "DATA". If the user so desires, a library of EDSF files can be set up in a single dataset.

3.1.1 Definition Part

The definition part of the EDSF serves to define the nature of the data structure which is stored in the data part.

To define a data structure, two things are needed:

1. A definition of the type of the data structure.
2. A definition of each of the fields with in the data structure.

Within the definition part of the EDSF the information that defines the type of the data structure is stored in a **index record** and the information as to the type of each field in the data structure is stored in a **field record**.

3.1.1.1 Index Records

The index record defines the type of a data structure. The following information is stored in it:

1. The name of the data structure.
2. The type of the data structure. The following types are defined:
   a) Singly linked lists. Each node in such a list has a pointer to the next node in the list.
   b) Doubly linked lists. Each node contains a pointer to both the next and the previous node in the list.
   c) Binary trees. Each node has two pointers in it to other nodes which are said to be the children of that node.

d) Networks. That is, any node can be connected to an arbitrary number of other nodes. Of course, linked lists and binary trees could be considered to be networks, but these types imply that special formats are to be used when they are displayed.

e) Arrays. The term array is taken to mean any collection of identically defined nodes which are accessed by means of one or more subscripts. The number of subscripts and their ranges are not defined in the EDSF. While the size of each element in the array will be defined by the fields in it, the total size of the array cannot be determined without knowing the ranges of the subscripts. However, when the extractor is extracting an array from a program, it has to know the amount of storage that the array occupies. To supply this information for the extractor, the size of the array is stored in the index record of an array.

f) Records. A record is defined to be one or more fields which are stored in adjacent memory. A simple variable is defined to be a record with one field.

3. The number of fields in each node of the data structure.

4. The total size of one node in the data structure. For an array, this is the size of each element in the array.

Each data structure (and therefore each index record) also has a data structure number associated with it. This number is an integer with the first data structure in the EDSF numbered one. This number is used to identify which data structure a pointer is pointing at. This data structure number is not stored in the index record itself but is stored as the key of the index record. This information could have been stored in the index record itself but NAM requires a key field for each record. While the NAM key does not have to be numeric, MANTES uses numeric keys for its records and I decided to remain compatible with MANTES by choosing numeric keys for the EDSF records.

3.1.1.2 Field Records

A field record is used to define a field of node. There is one field record for each field within the node. The following information is stored in a field record:

1. The name of the field.
2. The type of the field. This could be one of the following:
   a) Integer. A 4 byte 2's complement integer. This normally has values between -2^{31} and $2^{31}-1$.
   b) Short integer. A 2 byte 2's complement integer. This normally has values between -2^{15} and $2^{15}-1$.
   c) Real. A 8 byte real number.
   d) Short real. A 4 byte real number.

e) String. This is a varying length character string as defined in PL/I [IBM81c, P 23]. This consists of a 2 byte field giving the current length of the string followed by the characters that are part of the string.

f) Character. This is just an area containing a fixed length character string of arbitrary length.

g) Boolean. A 1 byte logical value. This storage is normally zero indicating false or not-zero indicating true.

h) Pointer. A pointer to another node in perhaps another data structure. If a field is a pointer the data structure number of the data structure that is pointed at is also stored. This means that untyped pointers (such as those used in PL/I) cannot be used with this system unless the user ensures that each pointer only points to the data structure that was specified.

3. The number of bytes that are required to store this field.

4. The displacement of this field within the data structure. Note that the displacement of the current field plus its length will not necessarily equal the displacement of the next field because of alignment considerations.

Much of the information stored in a field record is redundant. For example, the displacement of a field could be calculated knowing the size and displacement of the previous field and the alignment required by the current field. However, it was decided to store this extra information to save having to recalculate it each time it was required. Also, by storing this extra information, the EDSF can handle data structures in which the alignment of fields is not the normal alignment. This is the case when a structure is declared with the "UNALIGNED" attribute in PL/I [IBM81a P 22].

Field records also have to have keys. The field records in the EDSF should follow the index record to which they belong in the order which those fields occur in the node. As records in a NAM file are ordered by key, the keys of the field records have to be greater than the key index record to which they belong but less than the key of the next index record. To produce the key of a field record, it was assumed that no node would have more than 9999 fields. Making that assumption, the key could be defined as being RECORD-NUMBER + FIELD-NUMBER * 10^{-4}. Thus the key for a field record will be an real number greater than the key of the index record to which the field record belongs but less than the next integer.

If a data structure contains nodes which are identical to those of some other data structure, then a complete set of field records is not stored for the second data structure.

Instead, a special type of field record is used to indicate that the definition of this data structure can be found elsewhere. This record just contains the data structure number of the data structure in which the definition of the current node can be found.

3.1.1.3 Index Part Examples

To illustrate the various features provided in the EDSF, I will provide several examples. The first example is a set of PL/I declarations to illustrate some of the types allowed in the index and field records.

```
DECLARE  1  DATA_ARRAY (12),
            2  FIELD_1          FIXED BINARY (31),
            2  FIELD_2          FLOAT DECIMAL (6);

DECLARE  1  DATA,
            2  FIELD_1          FIXED BINARY (31),
            2  FIELD_2          FIXED BINARY (15),
            2  FIELD_3          FLOAT DECIMAL (13),
            2  FIELD_4          FLOAT DECIMAL (6),
            2  FIELD_5          CHARACTER (20) VARYING,
            2  FIELD_6          CHARACTER (20);
```

The following entries in the index part of the EDSF would be produced by the preceding declarations:

KEY	NAME	TYPE	# OF FIELDS	NODE SIZE	FIELD SIZE	OFFSET
1.0000	DATA_ARR	ARRAY(96)	2	8		
1.0001	FIELD_1	INTEGER			4	0
1.0002	FIELD_2	SH. REAL			4	4
2.0000	DATA	RECORD	6	62		
2.0001	FIELD_1	INTEGER			4	0
2.0002	FIELD_2	SH. INT			2	4
2.0003	FIELD_3	REAL			8	8
2.0004	FIELD_4	SH. REAL			4	16
2.0005	FIELD_5	VCH(20)[1]			22	20
2.0006	FIELD_6	CH (20)[2]			20	42

[1] Variable length character string. Maximum length 20 characters.

[2] Fixed length character string. Maximum length 20 characters.

Notice that between FIELD_2 and FIELD_3 of the record there are 2 unused (slack) bytes. This is because 8 byte floating point values are doubleword aligned.

PL/I does not have strongly typed pointers. This means that a pointer variable in Pl/I does not have to point an area of memory which is always defined to be of the same format. Thus, PL/I pointers cannot be represented properly in the EDSF as the index part of the EDSF assumes strongly typed pointers. However, Pascal pointers are strongly typed and the following example illustrates the use of the EDSF with pointers.

```
TYPE
   POINTITEM = @ INVITEM;
   INVITEM   = RECORD
                  NUMBER:        INTEGER;
                  QUANTITY:      INTEGER;
                  PRICE:         REAL;
                  LINK:          POINTITEM;

VAR
   TOP:          POINTITEM;
   CURR:         POINTITEM;
```

These declarations would produce the following EDSF entries:

KEY	NAME	TYPE	# OF FIELDS	NODE SIZE	FIELD SIZE	OFFSET
1.0000	INVITEM	RECORD	4	20		
1.0001	NUMBER	INTEGER			4	0
1.0002	QUANTITY	INTEGER			4	4
1.0003	PRICE	REAL			8	8
1.0004	LINK	PTR (1)[3]			4	16
2.0000	TOP	RECORD	1	4		
2.0001	TOP	PTR (1)			4	0
3.0000	CURR	RECORD	1	4		
3.0001	CURR	PTR (1)			4	0

[3] Pointer to storage whose format is defined by data structure number one.

Notice that there is no entry in the EDSF for the type POINTITEM. This is because POINTITEM just defines the type for a pointer variable. However, memory will undoubtedly be dynamically allocated for variables pointed at by the type

POINTITEM (i.e. storage of type INVITEM). Thus the type that POINTITEM points at (INVITEM) is present in the EDSF, as are all variables of type POINTITEM.

3.1.2 Data Part

The data part of the EDSF contains the current values that are stored in the data structure. It was decided to store this information in its internal form to minimize the work required of the extractor to extract a data structure. Thus the data part of the EDSF will essentially be a storage dump of the storage for each instance of a data structure. The address that the data structure occupied in memory is not stored in the EDSF at all. The only type of data item that cannot be stored in its internal form is a pointer. As the EDSF does not contain the absolute address from which a data structure was extracted, and pointers are normally implemented as absolute addresses, some conversion has to be done so that the structure can be reconstructed. To solve this problem pointers are stored as offsets within the data structure. The details of this conversion will be described when the data structure extractor is discussed.

For data structures that are stored in contiguous storage, such as arrays, there is no problem in determining the location and amount of storage to place in the data part of the EDSF. However, data structures such as linked lists may not be stored in contiguous storage. Instead of storing the

smallest block of storage that contains all of the nodes of a non-contiguous data structure, I decided to map a non-contiguous structure into a contiguous block of storage and then store that compressed form of the data structure in the EDSF. This operation will be performed on all data structures that are not necessarily stored in a contiguous fashion (such as linked lists, trees, and networks). However any contiguous data structure (such as an array or record) can be stored in the EDSF directly.

Within the data part of the EDSF, there are two possible approaches that could be followed for storing several instances of a data structure. Either the entire data structure could be stored in the data part for each instance of the data structure, or the initial state of the data structure could be stored in its entirety and for subsequent instances, the changes that have occured could be stored. I decided to use the the first method of storing the data because:

1. The extractor would be considerably simplified, and therefore would execute much more quickly.
2. If the user wished to see the ultimate state of the data structure, the system would not have to run through all previous states of the data structure. Each state of the data structure would be able to stand on its own.

A means is required to relate the records in the data part of the EDSF with the data structure to which they belong. This is done with the record key which indicates three things:

1. The instance of the data structure that is being dealt with.
2. The data structure to which this data belongs.
3. The order of the various records within a particular data structure.

The assumption was made that there would never be more than 10,000 data structures in the EDSF so the key can be encoded as INSTANCE * 10^4 + DATA-STRUCTURE-NUMBER + ORDER * 10^{-4}.

Within the data part of the EDSF there are 3 types of records: data records, no-change records, and absent records. They are described in the next 3 sections.

3.1.2.1 Data Records

Data records contain the data in the form described above. Normally one data record is used per node of the data structure. However, in some cases more than one record may have to be used per node as the maximum record length of a NAM record is 255 characters. The offset of the node in the data structure is also stored in each data record to aid in the reconstruction of the data structure.

3.1.2.2 No-change Records

In the data part of the EDSF, a no-change record indicates that the data structure has not changed since some previous instance of the data structure. Thus, instead of storing the entire data structure again, the instance where the data has been stored is recorded. This will save space in the EDSF.

The no-change record functions at the data structure level rather than at the node level. If only a minor change is made to the data structure the entire structure has to be stored again.

3.1.2.3 Absent Records

It may be that for a given instance, a data structure does not exist. This could be because the data structure is dynamically allocated and the statements that allocate it have not yet been executed. To indicate this fact, an absent record is stored in the data part of the EDSF for that instance.

Instead of having the absent record, there could just be no record for the data structure. However, it was felt that having a record would provide more security in case of a damaged file.

3.1.3 Limitations of the EDSF Design

There are several limitations in the current design of the EDSF. They are mainly related to the various types of data structure that can be defined in the index part of the EDSF.

The major limitation is that I provide no mechanism for having one data structure as a field of another. This manifests itself by the inability to define a substructure or an array within a structure (record). This deficiency was discussed at great length as the system was being developed and it was felt this would not provide any great restriction on the use of the system as a teaching tool as definitions can be repeated to obtain this effect.

Another fairly major limitation is the inability of the EDSF to support storage overlays. This would prevent the handling of Pascal's variant records and FORTRAN's equivalenced variables. This again is not seen as a major limitation to the current EDSF.

A minor limitation is the types of variables that can be defined in the index part of the EDSF. For example, there is no way of defining Pascal sets or enumerated types with the current form of the EDSF. However, if the lack of the above two types or any other type defined by a programming language proved to be a problem it probably would be simple to add that type to the EDSF.

Another minor limitation is the inability to store data structures that have to be at a specific address in memory (such as system control blocks). However, I feel that this will never become a major problem, given the purposes for which the system was defined.

Finally, the EDSF cannot handle dynamically sized arrays as supported by the PL/I statements:

```
SUBR:    PROCEDURE (X);
         DECLARE X(*)          FIXED BINARY (31);
```

These statements declare X to be an array of integers whose size is dependent upon the size of the argument of procedure SUBR. To support dynamically sized arrays, the total size of the array would have to be stored in the data part of the EDSF rather than the index part. As there is no way of doing this currently, dynamically sized arrays cannot be handled.

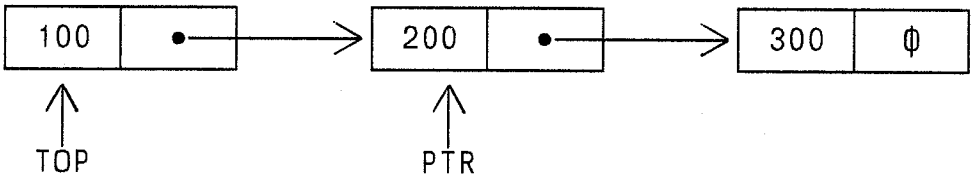
3.1.4 EDSF Examples

This section provides various examples of the complete EDSF. The first example shows the storage of a simple linked list which has two pointers pointing into it. The data structures are declared with the following Pascal statements:

```
TYPE
   AT_NODE = @ NODE;
   NODE    = RECORD
                DATA:          INTEGER;
                NEXT:          AT_NODE;
                END;

VAR
   TOP:    AT_NODE;
   PTR:    AT_NODE;
```

If the data structure is in the following state:



Then the contents of the EDSF will be as follows:

INDEX PART

KEY	NAME	TYPE	# OF FIELDS	NODE SIZE	FIELD SIZE	OFFSET
1.0000	NODE	RECORD	2	8		
1.0001	DATA	INTEGER			4	0
1.0002	NEXT	PTR (1)			4	4
2.0000	TOP	RECORD	1	4		
2.0001	TOP	PTR (1)			4	0
3.0000	PTR	RECORD	1	4		
3.0001	PTR	PTR (1)			4	0

DATA PART

KEY	OFFSET	DATA
10001.0001 10001.0002 10001.0003	0 8 16	100 ■ 8 200 ■ 16 300 ■ ϕ
10002.0001	0	0
10003.0001	0	8

In this example the character ■ is included to separate the various fields in the data. It is included in these examples to help the reader but is not present in the EDSF.

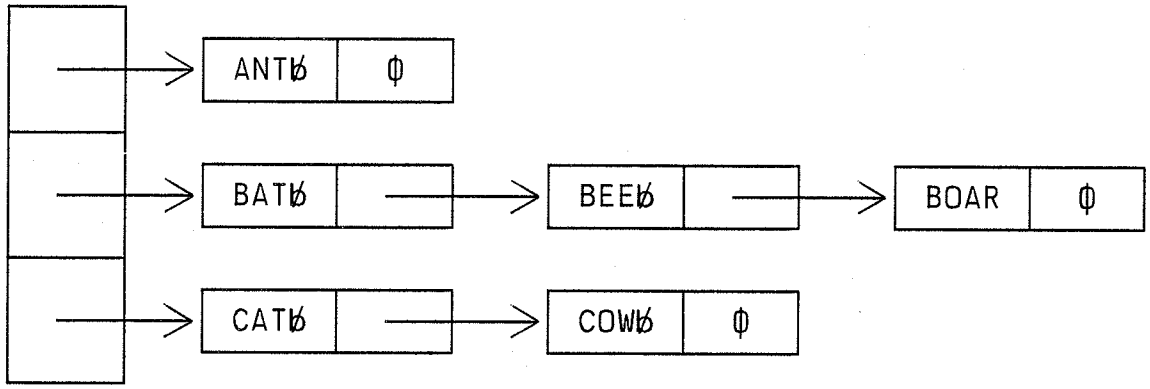
The pointer variables are converted to offsets within the data part of the EDSF. In these examples, the character ϕ is used to represent the null pointer. However in the data part of the EDSF, a special value is used. Pointer values in the EDSF will be discussed in more detail in section 3.2 of this thesis.

The next example illustrates the use of the EDSF with an array of pointers. Given the following Pascal declarations:

```
TYPE
   AT_NODE = @ NODE;
   NODE = RECORD
             WORD:  PACKED ARRAY [1..4] OF CHAR;
             NEXT:  AT_NODE
             END;

VAR
   HEADERS:          ARRAY [1..3] OF AT_NODE;
```

If the data structure currently was in the following state:



The EDSF would be the following:

INDEX PART

KEY	NAME	TYPE	# OF FIELDS	NODE SIZE	FIELD SIZE	OFFSET
1.0000	NODE	RECORD	2	8		
1.0001	WORD	CH (4)			4	0
1.0002	NEXT	PTR AT (1)			4	4
2.0000	HEADERS	ARRAY(12)	1	4		
2.0001	HEADERS	PTR AT (1)			4	0

DATA PART

KEY	OFFSET	DATA
10001.0001	0	ANTØ ■ Φ
10001.0002	8	BATØ ■ 24
10001.0003	16	CATØ ■ 32
10001.0004	24	BOAR ■ Φ
10001.0005	32	COWØ ■ Φ
10001.0006	40	BEEØ ■ 8
10002.0001	0	0
10002.0002	4	40
10002.0003	8	16

The order of the records in the data part of this file is only one of the many possible orders. The order would depend on the original location of each node in memory and the mapping used to place the data structure in contiguous storage.

3.2 THE DATA STRUCTURE EXTRACTOR

The job of the data structure extractor is to extract a data structure from a working program. It could involve just extracting the data structure once if that data structure is intended for use by the fetcher or extracting the data structure every time it is modified for use with the displayer.

Other systems for the extraction of information from a program while it is running (for example the BALSA system [BROWN84]) require that the user modify his program to generate the displays that he requires. This requires the user to identify the points where a change has been made to the data structure and invoke the extraction mechanism at that point. One of my main design requirements for the extractor was that there was some automated means of invoking the extractor at the correct points in the program. With such an automated facility, the system could be used by an inexperienced user as an aid to debugging. The BALSA system requires an "Animator" who will take a working program and modify it so that it displays the required information. Thus, the BALSA system could not be used easily by an inexperienced user as a debugging aid without a considerable delay.

Two schemes were considered for use in the extraction of a data structure from a program. They were the use of an interactive debugger to stop the program and extract the in-

formation, and the invocation of the extractor by a procedure call passing the data structure as a parameter.

3.2.1 Extraction via a Debugger

This was the scheme used in the Incense system [Myers 80] and there would be several advantages in using this scheme.

1. This would make the extractor totally transparent to the user. The system could set breakpoints via the debugger after any statement that modified the variable and then use some of the debugger routines to extract the information.
2. The extractor could also be combined with the display generator within the debugger allowing the user to set his own break points and then display the data structure only if necessary.

However, there would be one major problem with using the debugger to extract the information required: changes would have to be made at the source level of the debugger and that would make my system dependent on one specific version of the debugger. If any changes were made to the debugger, unforeseeable changes may have to be made to the extractor. The Incense system [Myers, 80] was developed at the Xerox Palo Alto Research Center for use with the language Mesa which was also developed there. The author of Incense had access to the source for Mesa and worked in close co-operation with the implementors of that language. However, there

was not a locally developed compiler/debugger available at this site and so this scheme was rejected.

3.2.2 Extraction via Procedure Call

In this scheme, the extractor would be invoked via an explicit procedure call after the data structure had been modified. This has the advantage that the extractor would not have to be written for a specific language or a specific version of a language. Any language which could call an external subroutine could then be used with the extractor.

One problem that would occur with a procedure call is the type checking that is present in most high level languages. It would be nice to invoke the extractor and pass as a parameter the data structure that is to be written to the EDSF. If the language from which the data structure is being extracted checks types on procedure calls, it would be very difficult to extract two different data structures from the same program. Indeed, one would have to make sure that the types matched between the declaration of the extractor and its use in the program. This would mean re-compiling the extractor every time it had to be used on a different type.

However there are usually ways around typing on procedure calls. Some ways that the type checking could be circumvented are:

1. In many versions of Pascal (for example Pascal/VS [IBM81b] and [IBM81c]) a variant record can be used to change a type. If the tag[2] field is not checked properly it is possible to change types quite easily this way. In other languages, there are often ways of doing this much more easily: for example PL/I [IBM81a] has the DEFINED attribute which allows an overlay to occur.
2. Type checking is often not performed well between separately compiled modules. That is, references to externally compiled routines are often resolved by a general system routine rather than something specific to the language being processed. IBM's linkage editor [IBM78] for example does not do any type checking on data items.

If the user desired to insert the procedure calls himself this would certainly be allowed. However, for a large program, the user would probably desire to use an automated mechanism to introduce the statements that would invoke the extractor. If the data structure is to be extracted every time that the a change is made, this requires a system that can identify that such a change has been made. I decided to do this via a preprocessor which will examine the users program, identify the statements that modify the data struc-

---

[2] The tag field is the field of the variant record which is used to indicate which of many possible definitions for the variant record is currently active.

tures that the user is interested in and insert the statements necessary to invoke the extractor. More details of the preprocessor that was written for this system are discussed in chapter four.

However, no matter which scheme is used to extract the data structure, I still needed a method of specifying which data structures are to be extracted from the program. A typical program will have many data structures in it and the user is probably only interested in a few of them. Also, if a user has indicated that he is interested in a certain data structure, he may not be interested in extracting each instance of the data structure. For example, if the user makes several changes to a linked list he may not want to save each change as it is made but rather the cumulative effect of those changes. Thus, I needed a way for the user to indicate his interest (or disinterest) in a certain set of statements. While this could have been done at the statement level, I decided to allow it only at the procedure level. This may mean that the user will extract the data structure from the program more frequently than he desired to if he is interested in only part of a procedure.

To specify which procedures to examine, the user is allowed to either include all specified procedures and exclude all others, or to exclude specified procedures and include all others. Two separate ways of specifying the procedures

to be examined are allowed in order to make things easier for the user. If only the ability to include specified procedures had been provided, then the user would have to name all procedures if he wished to extract the data structure every time it was changed anywhere in the program. The include/exclude facility only applies to the statements in the specified procedure; it does not apply to any procedures that may have been defined inside the specified procedure. Thus, if procedure "B" is defined inside procedure "A", then if procedure "A" is excluded, procedure "B" will still be included. If procedure "B" is to be excluded also, then it would have to be specified in the exclude statement along with procedure "A".

The user also has to specify the names of the data structures that he wishes to extract. This could have been done by just having the user provide the highest level name of the structure. However, this would mean that the preprocessor would have to interpret all of the declarations in the program to find the names of all of the fields of the structure and their associated types. It was felt that this would require a great deal of work to be duplicated between the compiler and the preprocessor. Also, if I desired to write a preprocessor for a second language, I would again have to deal with all of the idiosyncrasies that are present in the declarations for that language. Thus, I decided to define my own syntax of specifying the structure of the data structure to be extracted.

The syntax that I arrived at for describing a data structure is as follows:

```
<STRUCTURE> ::= <DS NO> <DS NAME> = <DS TYPE> <FIELDS>

<FIELDS> ::= <FIELD> | <FIELD> <FIELDS>

<FIELD> ::= <FIELD NAME> : <FIELD TYPE>
```

where:

<DS NO> is the number of the data structure in question. The data structure number is assigned by the user to the data structure. The data structure number is required so that pointers can refer to the correct data structure.

<DS NAME> is the name of the data structure. This is the name of the data structure in the program and that will be the name which is used in the EDSF.

<DS TYPE> is the type of the data structure. This would be one of the types allowed in an index record of the EDSF.

<FIELD NAME> is the name of a field of the structure. Again, this is the name from the program and that name will be used in the EDSF.

<FIELD TYPE> is the type of that field. This would be one of the types allowed in a field record of the EDSF.

This syntax gives a means of defining almost any data structure independently of the language in which that structure was originally defined.

For example, the data structures defined by the following Pascal/VS statements:

```
TYPE
   PTR     = @ NODE;
   NODE    = RECORD
                FIELD_1:        INTEGER;
                FIELD_2:        REAL;
                FIELD_3:        BOOLEAN;
                FIELD_4:        STRING (20);
                LINK:           PTR
                END;

VAR
   COUNTERS:       ARRAY [1..10] OF INTEGERS;
   POINT:          PTR;
```

would be defined with the following statements:

```
1  NODE      = RECORD
               FIELD_1:       INTEGER
               FIELD_2:       REAL
               FIELD_3:       BOOLEAN
               FIELD_4:       STRING (20)
               LINK:          POINTER TO 1

2  COUNT     = ARRAY (40)
               COUNTERS:      INTEGER

3  POINTER   = RECORD
               POINTER:       POINTER TO 1
```

Besides saving data structures that are being manipulated by the program, the user may also desire that the statement which modified the data structure be written to the EDSF as

well. This would allow the display of the actual statement along with the the data structure after that statement has been executed. In order to fit this in with the EDSF, it was decided that if the user gave a data structure name of "CURR_STMT", this would be interpreted to mean the previous statement. The user could then manipulate this information just as if it were a normal data structure.

Once the structure that is to be extracted has been identified, all that remains to be done is to copy it into the data part of the EDSF. For structures that are stored in a contiguous block of memory, this just involves copying that memory into the EDSF. However data structures that are not stored in a contiguous fashion, such as linked lists, pose more of a problem.

As has been mentioned previously, any data structure that is not stored in a contiguous fashion has to be mapped into a contiguous area of storage. To perform this mapping the following procedure is used:

1. All nodes of the data structure are identified and their addresses are stored in a table which is accessed by address.
2. Offsets are assigned to each of the nodes and these are stored in the table of addresses. The offsets are assigned so that the nodes do not change in position relative to each other. Thus, the node with the

smallest address has an offset of zero, the node with the next smallest address has an offset of the size of each node (subject to any alignment requirements that may exist for the node), and so on. The offset of each node is stored in the table of addresses.

3. The nodes of the data structure are copied to the data part of the EDSF. Any pointers that point to that data structure have their values looked up and the appropriate offset is substituted for the address.

However, a modification has to be made to this procedure to handle null pointers. In most programming languages, the null pointer is represented by an address of zero. However, an offset of zero cannot be used as the null pointer as that would be taken as a pointer to the first node in the data structure. Thus, whenever a pointer variable has the value of zero in it, a special null value (decimal -1 or hexadecimal FFFFFFFF) is stored in the EDSF.

Obviously, all of the pointers in the structure have to be followed in order to save all nodes that are part of that structure. However, sometimes pointers contain "bad" values. In other words, while the pointer does indeed contain the address of some location in storage, that storage does not contain what was expected. This could occur if the node at that location has been deallocated but the pointer has been left "dangling" or if the pointer field in the node has

not yet been initialized. While it is impossible to spot all bad pointers without access to the storage manager for that particular programming language, steps can be taken to ensure that some of the more obvious errors are detected.

On an IBM 370, pointers are normally stored as four byte values, however the largest possible value for an address requires only 3 bytes. Thus if a the high byte of a pointer is not zero then it is probably incorrect. However, even this simplistic scheme is not totally reliable as:

1. Often programmers use the high byte of a pointer for flags or other information.
2. IBM has recently changed it's hardware to support 31 bit addresses. Thus, the high byte of a pointer variable contains data.

Because of these two problems, very little pointer checking can be done for the current implementation of the extractor. However, in a future chapter, I will discuss various other schemes for checking of pointers.

3.3 THE DATA STRUCTURE FETCHER

The data structure fetcher takes a data structure that is stored in the EDSF and places it in memory. As the data part of the EDSF contains the data structure stored in internal form, fetching a data structure into memory will be a relativly simple task.

The parameters to the fetcher would be the name of the data structure to be fetched and the instance of that data structure that is required. If the user desired more than one data structure to be fetched, he would have to invoke the fetcher many times. The only case where the fetcher would bring more than one structure into memory is if the data structure that was requested contained pointers to other data structures. In this case, those other data structures would also be brought into memory by the fetcher.

In order to allow the fetcher to fetch more than one data structure and convert any pointers within the data structure from offsets to addresses correctly, I use a three stage procedure:

1. Determine which data structures are to be fetched. This will involve searching the index records in the definition part of the EDSF until the named data structure is located. Then, by checking the field records for that data structure, pointers to any other data structures can be located. This procedure is

repeated until all data structures that can be reached from the initial data structure have been located.

2. Allocate memory for each data structure. The amount of memory to allocate can be determined by examining the data part of the EDSF corresponding to the given data structure for data structures whose total size can be altered at execution time (such as linked lists) or the index part of the EDSF for fixed size data structures.
3. Read each data structure from the EDSF into the memory allocated for it in step 2 relocating each pointer by adding to it the load point of the data structure to which it points. If a pointer is encountered which has a value of hexadecimal FFFFFFFF then that pointer was originally null. Null pointers are assigned the value of zero.

The fetcher returns a pointer to the data structure. If a data structure is fetched that was originally allocated in a non-contiguous fashion, the pointer will point at the node that was stored at offset zero in the EDSF. However, that node may not necessarily be the first node in the data structure. To find the first node in the data structure, the user should request that a data structure that pointed to the first node be fetched. The fetcher will then fetch the pointer to the data structure and the data structure it-

self. The result of this request will be a pointer to a pointer to the data structure.

Chapter IV

IMPLEMENTATION

The system was implemented on the Amdahl 5850 (an IBM 370 compatable machine) at the University of Manitoba. The entire system (with the exception of the NAM interface routines) was written in Pascal/VS, IBM's Pascal compiler. Pascal was chosen as the implementation language as Pascal provides all of the necessary facilities required for this system but none of the complexity of PL/I. Also, Pascal has become the primary teaching language for this department.

When implementing the extractor, it was desirable to invoke it with a statement of the form:

```
EXTRACT (n, ID1, ID2, ID3, ..., IDn)
```

where n is the number of data structures that are to be saved and IDi are the names of the data structures. This would mean that the extractor would have to be declared to accept a variable number of parameters each one having a different name. It would be desirable to have the procedure header of the extractor take the form:

```
PROCEDURE EXTRACT (N:      INTEGER;
                   ID:     ARRAY [1..N] OF @ DATA_STRUCTURE);
```

This form of procedure statement is preferable as the various data structures can be accessed by subscripting one variable rather than by a unique name for each data structure. In fact, it was possible, with the use of an assembly language interface, to convert the parameter list from one form to the other. After this conversion had been done, there were no serious problems encountered in the implementation of the extractor.

A preprocessor was written to insert the procedure calls into Pascal/VS programs. These calls have to be inserted after every statement which modifies a variable that is part of a data structure that the user wishes to have extracted. In a Pascal program, there are two ways that a variable can be modified:

1. If that variable is the target of an assignment statement and
2. If that variable is passed as a "VAR" parameter to a subroutine.

If a variable that is part of a data structure that the user is interested in is the target of an assignment statement, it is easy to insert a procedure call after that assignment statement. However, if the variable is modified as the re-

sult of a subroutine call, the there is a problem in placing the call to the extractor as the next statement to be executed may not necessarily be the one immediately following the procedure call. Consider the following Pascal program:

```
PROGRAM TEST (INPUT, OUTPUT);

VAR
   X:   INTEGER; {This is the data structure
                  that the user is interested in }

FUNCTION FRED (VAR Y: INTEGER): BOOLEAN;
   BEGIN
      {Modify Y and return some result}
   END;

BEGIN
   X := 0;
   WHILE FRED (X) DO BEGIN
      {Some processing}
      END;
   END.
```

Obviously a call to the extractor will be inserted after the assignment statement which initializes X. However, X will be modified by the function FRED so a call to the extractor has to be executed immediatly after the function FRED returns. To ensure that the extractor is always invoked at the correct place, two calls to it would have to be inserted; one inside the body of the loop and one after the termination of the WHILE-loop. A similar problem also arises with the Pascal IF, CASE, FOR, and REPEAT statements. The Pascal REPEAT statement poses even more of a problem than the other statements as it may require the insertion of a

call to the extractor immediatly after the REPEAT; however, the need for the call to be inserted cannot be determined until the UNTIL clause is encountered.

To solve this problem fully, a two-pass preprocessor would have to be written: pass one would make a note of where the procedure calls should be and pass two would actually place them in the source. However, I decided not to handle this problem for several reasons:

1. The user can always get around the problem by coding:

```
PROGRAM TEST (INPUT, OUTPUT);

VAR
   TEMP:     BOOLEAN;
.
.
.
TEMP := FRED (X);
WHILE TEMP DO BEGIN
   .
   .
   .
   TEMP := FRED (X);
   END;
END.
```

2. The use of "VAR" type parameters is discouraged as these often cause side-effects [BATE82].
3. A two pass preprocessor would considerably more complicated than a one pass preprocessor and would only solve a problem which would not occur very frequently.

The Preprocessor was developed by constructing a LALR(1) grammar for Pascal/VS and then writing a parser with the help of Brent Beach's parser generator [BEACH75]. One problem that existed in writing the parser was that many Pascal statements require a single statement as a clause of the statement. For example, the WHILE statement only allows one statement in the body of the WHILE-loop. If the user wishes to have more than one statement in the body of the loop, he must use a compound statement which is indicated with a BEGIN-END block. If a call to the extractor has to be inserted after a statement which is the only statement in the body of a WHILE loop, then the preprocessor will have to convert that single statement into a compound statement. This would involve backing up to before the current statement to insert the BEGIN. In order to simplify the preprocessor, the assumption was made that BEGIN..END would enclose any section of code where a call to the extractor would be inserted. This insures that the preprocessor never has to convert a simple statement into a compound statement.

Two minor restrictions also had to be placed on the program that is the input to the preprocessor to make the grammar unambiguous. The places where these restrictions occur are:

1. The syntax for the case statement allows an optional semicolon after the last alternative but before the "OTHERWISE" clause or the "END". The semicolon is not allowed in the input to the preprocessor.

2. In a record, an optional semicolon is allowed after the fixed part of the record and before the end or the variant part. The preprocessor will indicate a syntax error if this semicolon is not present.

The preprocessor is by far the largest part of he system in terms of the number of lines of code required to implement it. This is because it contains two complete LALR(1) parsers: one to parse the data structure description statements and the other to parse the input program. However, the parse tables were all generated by a parser generator so implementing the preprocessor took little time. However, the fetcher and the extractor both involved some intricate code and so while they are shorter than the preprocessor they took a long time to implement.

Chapter V

FUTURE DEVELOPMENTS - THE DATA STRUCTURE DISPLAYER

The data structure displayer will take data structures which are stored in the EDSF and display them in a graphical form. There are several possible ways that the displayer could produce it's output: colour graphics screens, text screens, line printers, and plotting devices. The displayer should be able to produce output that can be processed on any of the above devices. Thus the displayer would be device independent and any device dependant code would be in one device specific module.

The major task of the displayer will be to lay out the data structure on the available area. There are two approaches that that could be taken. Either the system could lay out the entire display based on some internal rules, or the user could specify the position of every data item himself. The approach that I will follow will be a combination of the two techniques mentioned above. The user will specify which data structures are to be displayed, where they are to start, and in which direction they are to grow. Then, given this information, the system will decide where to lay out the rest of the data structure. It is felt that this approach will give the user the control over the layout that

is required without burdening him down with details. In order to lay out a data structure on the screen, the following information will have to be specified by the user:

- The name of the data structure.
- The start position of the data structure on the screen. This is specified in terms of a row-column pair.
- The direction in the data structure is to grow. This is one of UP, DOWN, LEFT, or RIGHT.
- The colour to be used when displaying this data structure. This parameter would be ignored if the display was directed to a non-colour output device.
- The field or fields of the node that are to be displayed. The user may not want to display all fields of a node as that could make the node take up too much space on the screen. In fact, for classroom use, the user will probably only want one field displayed.
- The type of pointer following to be done. Some pointers into data structures are used by the programmer to define the entire data structure; other pointers are used to identify specific nodes within the structure. If a pointer defines the entire data structure, the data structure should be displayed starting at that pointer. The various nodes of the data structure may inherit display information from the defining pointer such as their colour or their direction of growth. However, a pointer that is used by the programmer to

identify a single node will not have any effect on nodes other than the one that it points to. Thus, defining pointers can be used to assign attributes to the entire data structure; specifying pointers can assign attributes only to a specific node.

- Whether the name of the data structure is to be displayed or not.

For example, If the contents of the EDSF were as follows:

INDEX PART

KEY	NAME	TYPE	# OF FIELDS	NODE SIZE	FIELD SIZE	OFFSET
1.0000	NODE	RECORD	2	8		
1.0001	DATA	CH (1)			1	0
1.0002	NEXT	PTR (1)			4	4
2.0000	TOP	RECORD	1	4		
2.0001	TOP	PTR (1)			4	0
3.0000	PTR	RECORD	1	4		
3.0001	PTR	PTR (1)			4	0

DATA PART

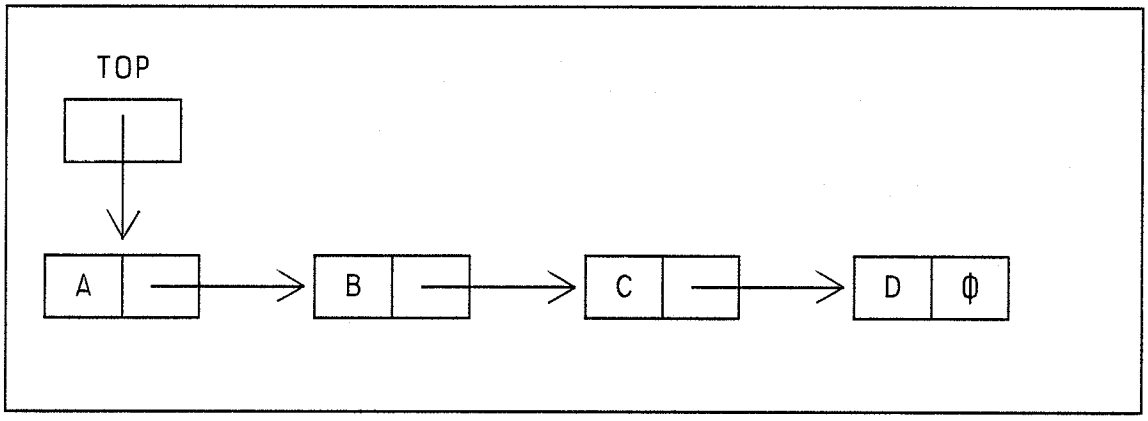
KEY	OFFSET	DATA
10001.0001	0	A ■ 8
10001.0002	8	B ■ 16
10001.0003	16	C ■ 24
10001.0004	24	D ■ ϕ
10002.0001	0	0
10003.0001	0	16

And the following layout information was specified:

```
DS NAME (NODE)
   START (2,1)
   GROW (RIGHT)
   DISPLAY FIELD (DATA)
   UNNAMED

DS NAME (TOP)
   START (1,1)
   DISPLAY FIELD (TOP)
   DEFINING
   NAMED
```

Then the following display would be generated:

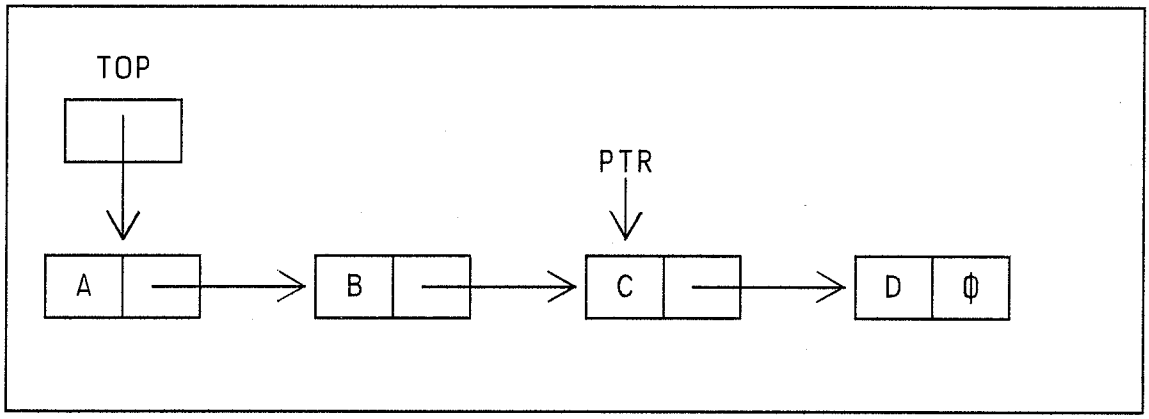


However, there may be circumstances when the user does not want the node to be at a fixed position on the screen. For instance, if the data structure being manipulated was a linked list which had a pointer into it, the user may only wish to know which node is being pointed at rather than seeing the storage for the pointer with an arrow leading to the node in question. In order to accomplish this, for pointer

variables, the user can specify a relative position to the node in question. If the following data was added to the layout information:

```
DS NAME (PTR)
   RELATIVE (ABOVE)
   SPECIFYING
   NAMED
```

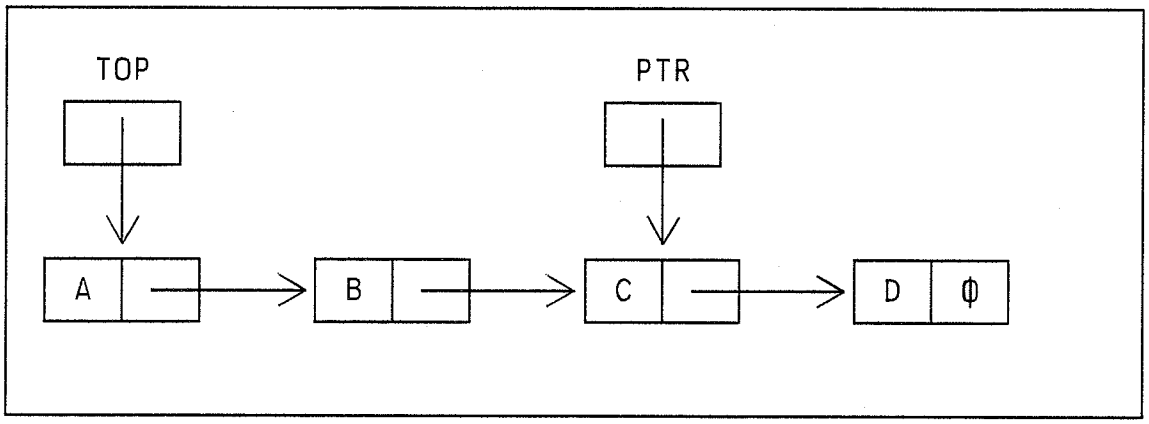
Then the display would look like:



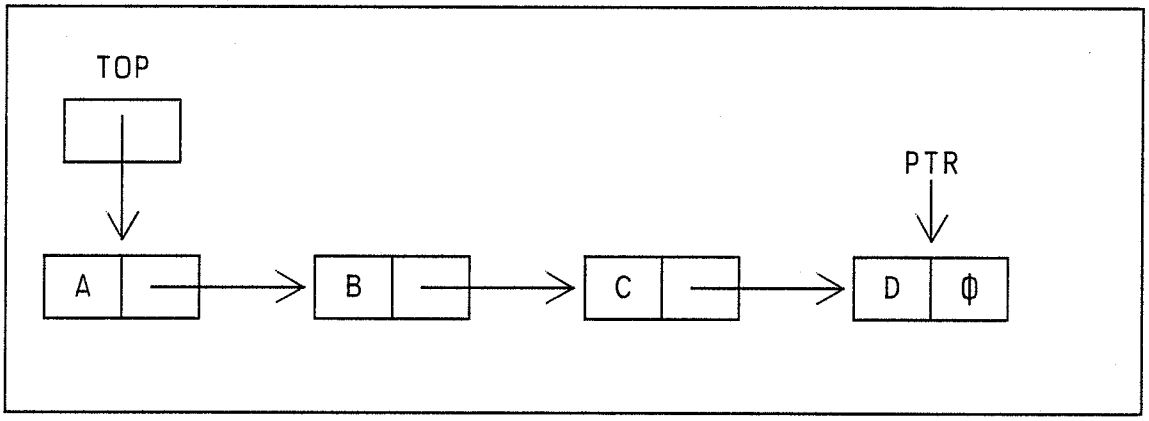
If however, the data structure PTR had been defined to be at a fixed screen location with the following specification instructions:

```
DS NAME (PTR)
   START (1,3)
   SPECIFYING
   NAMED
```

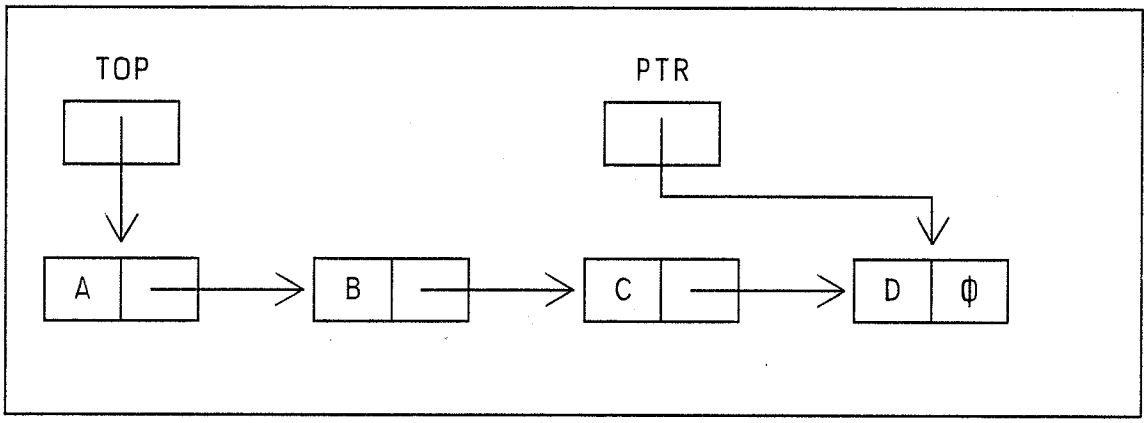
The results would look like this:



The advantage of allowing a relative position for a pointer becomes clear when another instance of the data structure is displayed. If PTR now points to node "D" rather than node "C" a display constructed with a relative pointer will look like:



However, the same data structure displayed with an absolute pointer rather than a relative pointer would produce:



If the data structure had many pointers into it and relative pointers were not implemented, the displayer would have trouble laying out the pointer lines. The user would also have difficulty following all of the various pointer lines around the display.

It is also possible to indicate the node that is being pointed at in another way: by altering its colour. This method doesn't clutter the display with extra lines and can be more eye-catching provided that suitably contrasting colours are used.

There are two ways of providing the position information to the displayer: either the information can be stored in a file or the user can be prompted for this information as the display is being constructed. In fact, the displayer uses both of these techniques in laying out the display. The

user will be allowed to specify the name of a file which contains the required information. This display information file is stored inside the data set which contains the EDSF. No special programs are provided for editing the display information as this can be done with the MANTES text editor. If the user on viewing the display realizes that there is a minor error in it, he can enter new information and have the display regenerated. The user will also be prompted if some required information is missing from the display information file.

One of the major problems to be faced in the design of the displayer is the time that is taken to change the display. For classroom use a delay of two or three seconds is the maximum allowable delay. Unless the display is connected to the computer by a very high speed link, the transmission of the data to the terminal will take more than three seconds if the computer tries to transmit the entire display for each instance. To solve this problem, I took into account the fact that most of the display remains unchanged from one instance to another. Therefore, it is only necessary to transmit the changes that have occured between the instances. This scheme could backfire however, if there is a radical change to the display (for example, if the user was displaying the data structure before and after a procedure call but not during the call). To insure that the displayer does not transmit modifications to the current dis-

play when in fact it should clear the display and start again, the displayer calculates the number of bytes that would have to be transmitted in both cases and then transmits the shorter string. This typically involves a great deal of computation in the displayer but gives a very desirable result.

Chapter VI

RESULTS

6.1 SUGGESTED IMPROVEMENTS

As with almost any system, there are always enhancements that could be made to many parts of the system. Some of the changes that have been suggested by both myself and others on examining the system have been:

1. Combining the extractor with a compiler to eliminate the need for a preprocessor to insert the calls to the extractor. This was discussed in section 3.2.1 of this thesis but was not implemented due to several problems, the most critical of them being the lack of a suitable Pascal compiler. Since that time, a Pascal compiler intended for student use has been developed at the University of Manitoba by H. Ferch. As we have access to both the source and the implementors of this compiler, we would feel no qualms about combining our system with that compiler.
2. Changing the method of detecting a change in a data structure. What is desired is the ability to invoke the extractor whenever a data structure is changed. Currently, changes to a data structure are determined at compile time and the appropriate statements are

inserted into the program. However, it is certainly possible for a clever programmer to fool the preprocessor by some underhanded coding. The proper time for detecting modifications to the data structure is at execution time, and the IBM system/370 provides a way of doing this through its Program Event Recording (PER) hardware. As described in [IBM76] the PER facility can be used to generate an interrupt every time storage between certain locations is modified. Thus, all that would have to be done is to have the compiler provide the location of the data structure within memory and then the extractor could be invoked whenever a change was made to that storage. However, the hardware support on the system/370 is not quite enough to allow this technique to be used properly. There would be the following problems:

a) The hardware only allows for the checking of one range of addresses. If the data structure was not stored in contiguous storage, or more than one data structure was being extracted the extractor would have to handle the invocations that would occur when storage that was not part of the data structure was modified.

b) Sometimes, a change that the programmer makes to the data structure may not actually occur in memory until some time after the change has apparently taken place. This could occur if a field of the

data structure was loaded into a register and manipulated there.

Thus, the IBM System/370 hardware does not provide quite the level of support that is required to implement this technique for the general case. This technique could be used to detect changes to a data structure stored in a contiguous area of storage. However, if a similar system was ever implemented on a different machine, this technique may become feasible.

3. Developing the system for use as a debugging tool rather than a teaching tool. The user should be allowed to make changes to the data structures as the program is running if the structures are not correct.
4. Adding all of the other Pascal types to the EDSF. This would not necessarily require major changes to the current EDSF format. For example, enumerated types could simply be inserted by defining a way of specifying each of the possible elements in the enumeration.
5. The current technique for detecting a null pointer is very elementary. As has been mentioned earlier, it would be much better to interface to the storage manager for the particular language being used. Then a pointer to a node which had been de-allocated could identified as such and the appropriate action taken.

In a system intended as a teaching aid, the detection of dangling pointers is very desirable as mistakes are often made in this area by students.

6.2 CONCLUSIONS

This thesis has described a file structure for storing data structures and various associated routines that operate on that file structure. The displayer will be functional by January 1985 so at that point, the EDSF can contain both test data for students programs and data to be displayed in class. The other parts of the system are currently functional but have not yet been classroom tested so I cannot comment on their use. Only time will tell if this system will prove to be an aid in teaching computer science.

BIBLIOGRAPHY

BATE82 Bate, J. A., Doyle, M. S., and Ferch, H. J. Computer Programming and Data Structures using Pascal, St Pierre: Charles Babbage Research Centre, 1982. P. 228-229.

BEACH75 Beach, B., A LALR(1) Parser Generator -- User's Guide. Department of Computer Science internal document.

BROWN84 Brown, Marc H. and Sedgewick, Robert. A System for Algorithm Animation. Computer Graphics, Vol. 18(3), July 1984, pp. 177-186.

FERCH82 Ferch, Howard J. Calling Sequences for NAM. Department of Computer Science internal document, 1982.

IBM76 IBM System/370 Principles of Operation. IBM Manual GA22-7000-4, 1976.

IBM78 OS/VS Linkage Editor and Loader. IBM Manual GA26-3813-5, 1978.

IBM81a OS and DOS PL/I Language Reference Manual. IBM Manual GC26-3977-0, September 1981.

IBM81b Pascal/VS Language Reference Manual. IBM Manual SH20-6168-1, April 1981.

IBM81c Pascal/VS Programmer's Guide. IBM Manual SH20-6162-1, April 1981.

MYERS80 Myers, Brad A. Displaying Data Structures for Interactive Debugging. Palo Alto: Xerox PARC CSL-80-7, June 1980.

Appendix A

NAM

NAM (Network Access Method) is the access method written to support the MANTES (Manitoba Text Editing System) editor at the University of Manitoba. While NAM can be used to access OS datasets (including PDS's), job output, and VSAM datasets, it also supports a special type of dataset for use by Mantes, known as a group.

A group contains a series of files arranged in a tree structure. Each of the leaves may contain any number of data records with the maximum size of those records being 255 characters. While the maximum size of a record is small it did not present a problem in the data part of the EDSF as it is always possible to use 2 or more records for each node rather than one. Each record also has associated with it a variable length key (in Mantes the key is the sequence number of the record).

NAM has 3 entry points which are used for the three types of operations that it allows. They are NIFILE which is used for operations on aggregates, NINODE which is used for operations on files within aggregates, and NIDATA which is used for manipulating records within files.

To specify record within a file within an aggregate NAM requires three pieces of information to identify the aggregate, the file, and the record being processed. This information is returned by NAM after every call to it and is either a 2 or 4 byte number. The aggregate identifier is known as a FID, the file identifier as an LNO, and the record identifier as a PNO. As it is the users responsibility to store these values and return the correct ones when required, it is possible to access several places in the same aggregate concurrently. In general, calls to NIFILE modify FID's, calls to NINODE modify LNO's and calls to NIDATA modify PNO's. There is no practical limit to the number of FID's, LNO's, and PNO's that can be active at one time.

The operations that can be performed by NAM are as follows:

1. Performed by NIFILE:
   a) **Open** a group.
   b) **Close** a group
   c) **Purge I/O operations.** This ensures that all changes that have been made to the group in memory are written to disk. This is done in to keep the disk copy of the group up to date.
2. Performed by NINODE:
   a) **Point** to a file.
   b) **Unpoint** to a file. This would be done when access to a file is no longer required.

c) **Create** a file.

d) **Delete** a file.

e) **Rename** a file.

f) **Move** a file to another position within the group.

g) **Add a password** to a file.

h) **Remove a password** from a file.

i) **Supply the password** to a file.

3. Performed by NIDATA:

a) **Point** to a record.

b) **Unpoint** to a record. This would be done to free a PNO.

c) **Insert** a record.

d) **Delete** a record.

e) **Read** a record. The record to be read can either be the current record, the previous record, or the next record in the file.

f) **Rewrite** a record.

g) **Change the key** of a record.

h) **Count** the records between two positions

i) **Compare** two records to see which one occurs first in the hierarchy.

After each of these operations, a return code is returned which indicates whether the request succeeded or failed. If the request failed, the value of the return code indicates the nature of the problem.

While NAM provides many operations as opposed to one or two for ordinary access methods, programs to process files using NAM are not overly complicated. For example, the pseudo-code required to print all of the records of the file FRED which is stored in the aggregate ROGERS.TEST.AGGR would be:

```
DECLARE
   FID, PNO, LNO, LINE

BEGIN
   NIFILE (open aggr, FID, 'ROGERS.TEST.AGGR');
   NINODE (point, FID, PNO, 'FRED');
   NIDATA (read current, FID, PNO, LNO, LINE);
   WHILE return code ≠ end of file DO;
      PRINT (LINE);
      NIDATA (read next, FID, PNO, LNO, LINE);
   END WHILE;
   NIFILE (close aggr, FID);
   END;
```

This code will function properly even if FRED is not a data file but is instead a directory file. If FRED is a directory, NAM will return all of the records in all data files that are below FRED in the tree.

NAM goes to great lengths to ensure that data stored in it's aggregates is protected in the case of system failure. If NAM detects that the last operation on a group did not complete normally, it will do a validity check on that group. The validity check makes sure that all of the internal pointers are correct. If it detects a problem, the er-

rant pointer is corrected or if the error is uncorrectable, an error message is produced.

While NAM provides many desirable features it is not widely used even at the University of Manitoba. This is because it was written for use by MANTES and as a result the user interface to NAM is not as "friendly" as that of other access methods. In fact, it is impossible to invoke NAM from any high level language due to NAM's non-standard method of accepting parameters. However, as all of my software is written in Pascal, I wrote low level interface routines to interface between NAM and Pascal/VS. All of the NAM interface code is in one module and is available for use by other systems.