DATA STRUCTURES FOR ALGEBRAIC MANIPULATION

by

J. G. Linders, M.A.Sc., F.B.C.S.

A Thesis Submitted to the University of London
for the degree of Doctor of Philosophy

UNIVERSITY OF LONDON
Imperial College of Science and Technology
1969-70

Preface

This thesis describes an approach to the manipulation of formal algebraic expressions by digital computer. It differs from others in that a major emphasis is placed on the ability to model and manipulate algebraic structure. The data structure used to represent a general algebraic expression not only contains the formal variables of the expression but also algebraic structural information as well as the map associated with the physical structure in storage. By having algebraic entities with more structure than is found in a character string, the language for formal algebraic manipulation is correspondingly simplified.

The algebraic data structure facility known by the acronym AIDS (Algebraic Interpretive Data Structures) as well as the supporting macros and manipulative routines provide a base from which to construct compilers for algebraic symbol manipulation. In this thesis AIDS is used to construct a simple command language for algebraic manipulation (CLAM). A number of essentially trivial examples are given to demonstrate the capability of AIDS.

AIDS has been written for a multi-access type of environment. All routines are hence re-entrant so that they may be shared by several users simultaneously.

The first part of the thesis discusses the

development of data structures and languages for algebraic manipulation. The syntax and operation of AIDS is then described. The system is summarised in the appendices.

The present implementation has been written for the IBM 360 although the definition is essentially independent of the hardware.

Table of Contents

	Page
Preface	i
Table of Contents	iii
Acknowledgements	vi
Chapter I	
Introduction	1
Chapter II	
Algebraic Manipulation Schemes	4
Symbolic Manipulation in High Level Languages	7
Polynomial Manipulation Systems	9
General Algebraic Systems	10
Requirements for Algebraic Manipulation	12
Chapter III	
Data Structure Schemes	17
Data Structure Types	
1. Arrays	17
2. List Structure	18
3. Trees	19
Data Structure for Algebraic Manipulation	26
Summary of Data Structure Requirements	30
Chapter IV	
Algebraic Interpretative Data Structure	32
General Philosophy of AIDS	32
Description of AIDS Data Structures	36
External Representation	37
Syntax of Algebraic Expression	37
Basic Symbols	37
Identifiers	38
Numbers	38
Variables	39
Function Designator	39
Algebraic Expressions	40
Internal Representation	40
Algebraic Values	41
Further Syntactic Entities	41
Representation of Algebraic Structural Information	44
Representation of Exponents for Composite Elements	45
Function Designator Representation	47
Implementation Restrictions	49

Organisation of Elements Within a Structure 50
Canonical Form of Data Structure 50
Extended Form of Data Structure 51
Further Extensions 58
Accessing Elements Within the Data Structure 59
Conversions Between External and Internal Representations 62
Operation of the Syntax Analyser 62
Syntax Analyser Conventions 66
Forming Algebraic Structures 68
Conversions Between Data Structure and External Representations 69
Recursive Facility in AIDS 70
Description of Stack 71
Recursive Programming Macros 73
Data Stacks and Data Stack Operations 77
Data Management 79
User Data Area 79
Stacks 80
Free Storage Scheme 81
Secondary Storage Facilities 84
Catalogue Facilities 85

Chapter V
Algebraic Operations 86
Data Transmission 90
Arithmetic Operations 92
Rational Arithmetic 92
Symbolic Addition and Subtraction 94
Multiplication and Division 95
Logical Functions 100
Equivalence of Simple Elements 100
Equivalence of Composite Elements 101
Functions 103
Numerical Evaluation 104
Removal of Parentheses 105
Factoring 106

Chapter VI
Algebraic Simplification and Substitution
Simplification 107
Simplification in AIDS 114
Simplification of Simple Algebraic Data Elements 115
Simplification of Composite Algebraic Data Elements 118
Term Simplification 119
Simplification of Simple Algebraic Expressions 120

Resolving Structural Complexity 121
Substitution 124

Chapter VII
Differentiation and Integration 126
Differentiation and Integration in AIDS
Differentiation 128
Integration 132

Chapter VIII
CLAM Algebraic Interpreter 133
Description of CLAM 133

Chapter IX
Summary 137
Future Enhancements 137
References 141
Appendix I - Syntax for Algebraic Expressions 144
Appendix II - Summary of AIDS Macros 148
Appendix III- Control Word Formats 157
Appendix IV - Description of Condition Byte after Logical Tests 158
Appendix V - Table Formats 159
Appendix VI - Summary of CLAM Commands 160

Chapter I

Introduction

The advent of the commercially available digital computer was to herald not only a new electronics orientated technology but the start and development of disciplines concerned with the efficient use of digital computing equipment. The initial interest in digital computers was little more than to emulate and automate the functions of a desk calculator, however with the exploitation of its arithmetic computational ability interest was soon focussed on its potential as a tool for non-numeric processing.

The digital computer can now be thought of more as an information processing device rather than as an automatic calculator. As hardware becomes faster and more sophisticated and with a major emphasis on storage media and access techniques, the functional aspects of computer usage are rapidly enlarging. Hardware development to date has been concentrated on extending the capability of the computer as a system through the development of special purpose peripheral devices such as visual displays.

The structural organization of the CPU and main memory has remained essentially unchanged and is as proposed by von Neumann in the early 1950's with the only structuring among the cells of main

memory being the implicit ordering of the natural numbers used for addressing. Recent experimental machines deviating from the von Neumann concept (refs. 1,2) have been constructed with the express purpose of simplifying the functional operation of a computer. These machines essentially use a non-linear addressing scheme which is realised by a hardware mapping on to a linear store. Paging, segmentation and some stack machines provide another variant of program addressing structure realised through extensive hardware and software systems. The need for these extended addressing schemes has been essentially to cater to a large dynamic environment.

The emphasis has not only been to add more hardware for addressing but also to extend the functional capability of the CPU by microprogramming standard sequences of instructions as single operations. The development of inexpensive integrated circuit modules suggests that microprogramming could become a very powerful facility to simplify machine usage by permitting more complex operations.

The linear machine with a standard instruction order code lends itself to realising algorithms for numeric computation. Lists and multi-dimensional arrays of values are readily stored and accessed on a linear store. The organisation of data elements

remains the same throughout the life of the program for numeric calculations. As shall be seen structures for algebraic manipulation are inherently dynamic and put more demands on the system for economic handling both in accessing and storage requirements. However if structural entities and operations on these entities can be defined, access and manipulation could be realised through hardware. With this in mind the development of algebraic manipulation on a digital computer and data structures is reviewed in Chapters II and III in an attempt to recognise essential processing functions as well as data representations and organisation.

Chapter II

Algebraic Manipulation Schemes

Development

Considerable progress has been made in recent years in the area of algebraic manipulation. Many schemes have been reported with, as of yet, little or no duplication of effort. Each scheme tends to be unique in some important aspect such as function, definition, efficiency, capability etc. Each in turn can be characterised by data base, functional capability, and method of operation. Many schemes are limited to specific algebraic functions such as differentiation or integration. An exhaustive survey up to August 1966 is given in references 16 and 17.

The first schemes developed were specifically designed for symbolic differentiation. The input was rather crude and closely resembled the internal representation (Ref. 4). Later schemes (Ref. 5) were developed more specifically to differentiate FORTRAN-like expressions.

Integration schemes (Refs. 5, 6) constitute the next level of development. The SAINT system for symbolic integration was a very significant development employing heuristic techniques in LISP. This system was capable of solving calculus problems with a high degree of success. The author further claimed that 100% of the problems having solutions

could have been dealt with with some minor changes to the system. A more ambitious scheme is that of Moses (Ref. 8), however it appears that further development along this line will require some form of man machine interaction system.

Polynomial manipulation systems were the next major area of development. These systems are confined to classes of formulae for which efficient algorithms can be implemented and include polynomials in one or more variables, rational functions, power series, trigonometric series and other series, etc. ALPAK (Refs. 9, 11) was the forerunner of such systems with further sophistication introduced in ALTRAN (Ref. 13) and the PM system (Ref. 12).

Another group of systems is designed to deal with the secondary school type algebra problems and basic problems in calculus and differential equations. The data base is a class of well-formed formulae generated from variables, numbers, arithmetic operators, the differentiation operator and a special function facility. The two most notable examples of this class are FORMAC (Refs. 14, 15) and SYMBAL (Refs. 18, 19).

Further sophistication is achieved in more general systems which permit the definition of a data base and in some cases the operators involved. These systems (Refs. 20-23) tend to be more experimental, sacrificing efficiency for sophistication.

SYMBAL (Ref.19) is the only algebraic system yet produced which represents a formal and generalised approach to symbol manipulation. The language is a semantic and syntactic generalisation of ALGOL 60 along the lines of EULER (Ref.27). From ALGOL it inherits the concept of program block structure and the conditional and GO TO statements. It uses the full recursive facilities of ALGOL in executing some of its functions as well as providing the same recursive facilities for user programs. From EULER it has taken the concept of having only a single declaration "NEW" for introducing names, while the values and types are handled dynamically. The initial value of a variable is its name taken as a string. This may dynamically take on other values by assignment.

A special list data structure is used to represent algebraic expressions. A new data type "vector" replaces the array of ALGOL. A vector is of variable length and can be assigned to single variables.

A SYMBAL program is essentially ALGOL-like. A well formed expression may contain the following operators:

1. arithmetic
2. relational
3. logical
4. equality

The basic syntactic entities include numbers, variables, labels, and functions. The types of values which are defined in the syntax include: 1) undefined, 2) algebraic, 3) logical, 4) label, 5) vector, 6) string, and 7) procedure.

Immediately after its declaration, a simple variable has the status "atomic" and is of type "undefined". An expression in SYMBAL takes the same form as an expression in ALGOL, being defined by essentially the same syntax. Simplification is also implicit in SYMBAL but is dependent upon mode values set by the user.

The flexibility of its list data structures provides SYMBAL with much of its capability.

Symbolic Manipulation in High Level Languages

A symbol manipulation capability exists in many high level languages. Such languages invariably have a data type "STRING" on which procedures can be defined for pattern matching and replacement.

SNOBOL (Ref.24) is a string manipulation language which runs as an interpreter. The primative data structure is a string which can be used to build up more complex tree structures. Because it is interpretive a high overhead is incurred wherever it is used to write other algebraic interpreters. Implementation constraints restrict the representation of numbers and the size of structure is limited to the available work space.

LISP (Ref.25), of course, provides a powerful programming tool for symbolic manipulation through its recursive and function definition facilities. It too is interpretive and incurs a sometimes unjustifiable overhead when used to write other programming systems. It is unfortunately unwieldy

in its use of parenthesis and suffers most in garbage collection when the main store must be re-organised. Existing data structures in the main store cannot be readily removed to backing store when more space is required in the main store.

PL/1, in attempting to incorporate all desirable programming features into a single language can also be adapted to symbolic processing. Dynamic string facilities exist in the form of variable length strings but this is wasteful of store usage as the maximum length string, defined by the user, is always allocated. Powerful list processing facilities are available in PL/1, however, until recently the implementation of these facilities has been extremely unreliable. PL/1 also maintains its own local map attached to all structures which tends to burden any user defined system with a further unnecessary overhead.

Many other languages are of course adaptable for symbolic processing, however their facilities are more useful in evolving concepts rather than in producing an economical algebraic system. As symbolic manipulation can make excessive demands on both storage requirements and processor cycles, it is important to consider schemes which will tend to minimise both of these.

Initially algebraic systems were developed almost exclusively for batch-type operations. The trend now is to develop interactive systems in which the user can direct the system operations for increased simplicity and efficiency.

The capability and operation of each class of algebraic system is most readily appreciated by discussing representative systems from each class.

Polynomial Manipulation Systems

The ALPAK system for polynomial manipulation was written at the Bell Laboratories for the IBM 7090. The initial version defined polynomial arithmetic on its pre-defined data structure elements. The operations provided through subroutines and macros include addition, subtraction, multiplication, division, differentiation of terms, zero test, non-zero test and an equality test on symbolic elements. The user was required to understand many of the intimate details of the system. The organisation of variables within a term is defined by the user in a special format statement which is stored for run-time use. The input of coefficients and exponent values is on a term by term basis according to the predefined format.

It was a natural step from a polynomial manipulation scheme to a rational function facility. A rational function is represented as an ordered pair of polynomials, namely its numerator and denominator respectively. These are stored in the polynomial canonical form and are relatively prime.

ALPAK provides a greatest common divisor of polynomials in several variables so that each rational function can always be stored in its canonical format.

ALPAK was further extended to permit solving by Gaussian elimination systems of equations linear in certain variables and with coefficients which are rational functions of other variables. A limited facility for substitution was also introduced. Essentially all ALPAK programs resemble assembler programming in FAP macros. The greatest limitation of ALPAK is that it can only continue until there is no further work space in main store. ALPAK has been used to solve problems in queueing theory, astonomy and wave propogation in crystals to name but a few. Even though it lacks the elegance of a concise command language it pioneered the way for special purpose algebraic systems.

General Algebraic Systems

FORMAC and SYMBAL are both symbolic processors, each associated with a well known high level language. Whereas SYMBAL is a complete system in its own right, FORMAC is essentially a preprocessor for FORTRAN IV, translating symbolic requirements into FORTRAN calls to a set of special object time routines.

The FORMAC programming language is a proper extension to FORTRAN IV and consists of the full FORTRAN IV language plus 4 further declarative statements and 15 executable statements for symbolic processing. In addition it introduces

symbol manipulation operators from which symbolic expressions can be created for manipulation at object time. Decisions based on symbolic expressions generated at run time can be used in the logic of program control during execution.

The operator set of FORMAC includes:

1. arithmetic operators - , +, *, /, **
2. special operators - FAC (factorial), DFAC (double factorial), COMB (combinatorial), DIF (differentiation)
3. the trigonometric function - EXP (exponential), LOG (natural logarithm), SIN, COS, ATAN, TANH

FORMAC has its own internal representation for symbolic expressions (see chapter III). Simplification is performed automatically after all symbol processing operations as described in chapter VI.

FORMAC was designed primarily for the batch processing environment to provide a symbol manipulation capability for FORTRAN IV programs. As with FORTRAN it is essentially simple and does not provide, for example, recursive facilities. Further it is a rigid system not readily adaptable to special user requirements.

All expressions must be contained within the available work space and substitutions are made for symbolic variables wherever possible. A trigonometric function is evaluated to a numeric result whenever possible.

Requirements for Algebraic Manipulation

From the previous discussions it is possible to identify certain basic requirements for formal algebraic manipulation systems. The realisation of the concepts is usually associated with an overall systems philosophy. Ideally, of course, it is desirable to have sufficient generality in the concepts to permit flexibility for achieving future enhancements. At the same time, it is necessary to identify a limited number of basic primitives which can be combined in a logically consistent manner to produce unambiguous constructions.

The basic requirement for algebraic manipulation is of course a suitable data base. The data base is often related to the type of problem being solved. Essentially it is necessary to provide a data structure whose elements can be used to model an algebraic expression. The choice of data base is also associated with other considerations such as mobility, accessing and referencing of data elements, as well as processing and storage efficiency. As algebraic data structures tend to grow very large any general scheme should include provision for moving complete data structures or their elements to backing store. Equally important is the ability to cope with the dynamic data environment of algebraic manipulation through schemes which readily permit

dynamic extension and modification of the data structure.

It is necessary to identify the primitive operations of algebra and consider their implementation on the data base. Each operator must affect the elements of the data base in accordance with predefined schemes. The basic operations which are essential include arithmetic (addition, subtraction, multiplication and division) and tests on data elements for total or partial equivalence. These operators are defined on symbolic elements with arithmetic on numeric data items constituting the degenerate case. The basic algebraic operations are required to realise more complex algebraic functions such as symbolic differentiation and integration.

Simplification is also an essential process in any formal algebraic scheme in that it permits the removal of redundant data. It is usually associated with operating efficiency and as such is an important criterion in the design of an algebraic system. Simplification must be an integral part of on overall system philosophy.

Symbolic processing is not an end in itself. Often it is necessary to associate values with the symbolic variables and numerically evaluate an expression. This can either be done by replacing all symbolic values by numeric data and evaluating the resulting reduced arithmetic expression. Alternatively, it may be desired to maintain the symbolic representation of an expression and perform the evaluation by another procedure which associates numeric value with all symbolic data items.

There are many algebraic procedures which are desirable in such systems. For example, it is often necessary to perform such operations as removal of parentheses (expansion) and the inverse operation, namely, factoring. The former is an exact process while the latter can sometimes lead to many equivalent results and hence usually requires user defined constraints or direction.

A function facility for defining run-time relations is an analogous facility except that it is more directly under user control. Such a facility can, for example, be used for defining in-line substitutions through side relations. This is over and above a function definition facility within algebraic expression(e.g. representation of a derivative).

The essential requirement for a general algebraic system can be summarised as combining a number of basic primative algebraic operations on a flexible data base. These facilities must be the basis on which to construct specific algebraic systems.

Chapter III

Data Structure Schemes

The usefulness of a data processing system can often be judged by the mechanisms through which it stores, references and manipulates data. The basic data handling capability is associated with the hardware order code and involves specific operations for pre-defined data representations. Further capability is achieved by imposing a software hierarchy of manipulative processes on the hardware based facilities.

Data is the representation of information. For digital computer applications the internal representations are invariably associated with some binary code. Within any process data can be distinguished as being of one of the following types:

1. Instructions - active data elements
2. Values - passive data elements manipulated by the active elements
3. Control - passive elements used by active elements for program logic and control

Data values constitute primitive data elements which can be organized into meaningful collections, called data structures for purposes of referencing, accessing and manipulation. A data structure in a programming language consists of three main parts:

1. A notation in the source language for referencing and manipulating the data elements in the structure.

2. An internal organisation scheme for the data elements
3. A mapping algorithm for relating the references in the external notation to the main store locations.

The internal representation of data elements is dependent upon the choice of computer. Each atomic data element has associated with it attributes which are usually implicitly known to the processing procedure. In some cases the attribute values are encoded within the data elements themselves as in IPL (ref 26). Each data element is ultimately referenced by an absolute machine address.

The genesis and evolution of data structures has been closely associated with both application areas and programming languages. More sophisticated data structuring schemes evolved with the development of more powerful and flexible programming languages.

Data Structure Types

1. Arrays

The simplest type of data structure to implement and reference is the one-dimensional array, or vector, implemented as a block of contiguous words in memory. All elements in the structure have the same attributes. The mapping algorithm references each element as an offset from the base of the structure. This structure is exemplified by the one-dimensional array in FORTRAN.

A character string is also a one-dimensional structure. For a byte or character machine, each character occupies one byte and is referenced in much the same manner as an array element in FORTRAN. If a character string is mapped onto a word machine, the mapping algorithm must determine in which word and part thereof the referenced character lies. Similarly, a table is another instance of a one-dimensional structure.

The only structure associated with the one-dimensional array is a single level ranking of elements according to position. This type of organisation is usually static in extent.

Multi-dimensional arrays are an extension of the vector concept to several dimensions. Each element in an n-dimensional array is referenced by means of n subscripts (S_1, S_{2_1}, ... S_n). The mapping algorithm for any element is

$$\text{element} = \text{base} + S_1 - 1 + \sum_{I=1}^{n-1} (S_{I+1} - 1) \prod_{J=1}^{I} D_J$$

where (D_1, D_2, ... D_n) is defined to be the extent of each array bound.

All elements again possess identical attributes. Even though all the bounds need not be identical a multi-dimensional array is usually wasteful of space when several of the elements do not exist.

An n-dimensional array is usually implemented as a static structure which has n degrees of ranking for the elements.

2. List Structure

A list structure is characterised by the explicit links connecting the data elements. The data elements are chained together with either single or double pointers for forward or bi-directional referencing respectively. Each element is stored as a contiguous block according to some pre-defined format.

A list element may be either atomic, in which case it can be considered a primative, or it may reference other list elements. In this way, complex data structures are built up in which the component elements may have differing characteristics. The topology of such structures in the most general form is a graph.

List structures are particularly well suited to an environment in which the number of elements can change dynamically and new elements are dynamically created. Unlike most other data structure schemes the organisation of the elements may be changed dynamically permitting the creation of complex structures. Because list structures are pointer based the size of the overall structure is limited by the addressable space available. Garbage collection can be a very serious problem especially as common elements can be

referenced from several other elements. An element can only be moved (or removed) when all pointers to it have been appropriately adjusted. Access to elements in a list is only possible on a sequential basis and the explicit chaining of elements together requires substantially more storage than when they are represented as a group of contiguous locations. Security can only be provided at run time by checking data types interpretively and not at compile time.

The programming languages LISP (Ref.25) and IPL (Ref.26) which use lists as their data structures provide a conceptional economy and elegance not readily found in other programming languages. List structures are also provided in PL/1 using the BASED facilities within the language.

3. Trees

A tree data structure is a directed graph in which each element, except the first, is uniquely addressed through a higher level element. A data element in the tree may consist of sub elements, each of which in turn may be further decomposed to any level. The terminal elements are unique primitives which may have differing characteristics.

The outstanding examples of tree implementation are the data structures of COBOL and PL/1, record classes and code word schemes. Their differences are derived mainly from their use in programming schemes.

The data structures of COBOL and the basic data structures of PL/1 are both definitions of multi-level trees. The data structure of COBOL provides a format description of the fields within a given record type where the variables reference the fields of the current record in the working store. For PL/1 there may be many instances of the same data structure currently active (e.g. it is possible to have an array of structures), however each has a unique identification.

Each element in a tree structure has its own set of attributes. These attributes are known only to the compiler in both COBOL and PL/1 for subsequent references and in the case of PL/1, for data conversions.

For COBOL the data structure description provides a mask for interpreting and referencing a given area of store. The mapping is built into the object code. However, for PL/1 the map associated with the structure is stored with the data in a "structure dope vector". Each substructure has its own dope vector for referencing its elements. A program reference to an element in a structure requires sufficient name qualification for proper referencing of the associated dope vectors. The dope vector organisation is dependent upon the stucture element type (e.g. array, string, structures, etc) and includes bounds values, offsets and lengths etc.

The basic data structures of PL/1 and the data structure of COBOL are both static in organisation and extent. These are well suited to such environments as found in commercial data processing where file structures are constant and hence no dynamic structuring is required. However, there are dynamic environments such as computer-aided-design, algebraic manipulation and dynamic modeling where a dynamic tree data structure is at least desirable if not essential.

A dynamic tree can be defined in PL/1 either through the use of pointers with the BASED facilities or to a limited degree with self defining data. A self defining record is one which contains, within itself, information about its own fields, such as length of string or number of elements in an array. In the former case a list tree structure is created while for the latter case a BASED structure is declared to have either one adjustable array bound or one adjustable string length, governed by a variable contained within the structure itself. This variable is assigned a value from a variable outside the structure when the structure is allocated. This facility is rather limited in that when specifying adjustable data such as an array bound, the bound must be the upper bound of the leading dimension of the element with which it is used. The dimension must further belong to the last element in the structure declaration, or to a minor structure containing the last element.

The conception, design, and use of record classes was pioneered in AED-O (Ref.28) and extended in AED-1. Records in AED are known as beads or n-component elements while the term plex is used to denote a group of interrelated records linked by references. There are no record class declarations as such in AED. The components of a bead are each declared independently with their offset within the bead specified as an integer constant. Reference fields are declared to be of type INTEGER while other components can be of type REAL, BOOLEAN or INTEGER.

Record handling as proposed by Hoare and Wirth is a refinement and formal generalisation of the concepts as found in AED. The proposal is for an extension to existing languages such as ALGOL 60. (Ref. 29)

In Record Handling the objects of a computational model are divided into a number of mutually exclusive classes. Each class is described by a record class declaration which denotes the attributes associated with all objects in the class. Each instance of an object in a given class requires the allocation of a fixed block of store for the record and the assignment of a value to each attribute field. In order to uniquely identify a field in a particular record the programmer must both name the field by its identifier and also indicate the name of the class to which the record belongs by a construction known as a field designator.

Records within a class can be referenced either by reference variables or by a state variable associated with each record class which always points to any one record within the class.

The use of reference variables in Record Handling provides the mechanism by which complex dynamic structures can be created in terms of well defined components. Within both AED and Record Handling new records can be created dynamically and associated with existing records within the system. A structure grows by establishing explicit links with new elements. The accessing and storage overheads are reduced considerably by the implicit structure of the n-component elements. Further it is possible to perform compile-time checks for data type, etc. The use of pointers for references still limits the extent of a structure to the addressable space available.

Another form of dynamic tree is the codeword scheme as first proposed by Iliffe and Jodeit (Ref.30). The two most notable implementations of this are the BLM (Basic Language Machine of ICL) and the data structure of ICES (Ref.31). A Codeword structure is built up of linear sequences of elements which may themselves be code words. Each codeword defines the address of a block of data, (which may be numeric data, program instructions or a set of codewords), as well as its length, type and other accessing information. The first set of control words is called the process base. The mechanics of the system are transparent to the user who communicates

with the system through a basic programming language (Ref.31) which defines basic primative programming operations on the codewords and their elements.

ICES was designed to cater to the dynamic aspects of engineering design. It provides facilities for both dynamic array capability and a relational (record handling type) data structure. The dynamic array capability is an extension of FORTRAN arrays implemented by using a codeword scheme. Dynamic arrays are known to the system through explicit declaration, all other arrays being FORTRAN type dimensioned arrays. The dynamic arrays are segmented into component elements so that allocation of space in working store can be made when the data is referenced through the use of a single level store concept. Through this facility it is possible to build up and manipulate dynamic trees to any level in which all of the elements are of the same type.

The overhead to reference an element in the main store is marginally greater than that associated with a dimensioned array although the overhead for retrieving data segments from secondary store can be appreciable. An array can grow dynamically either by redefinition or in terms of a fixed increment size.

ICES also has a facility for modelling a dynamic tree in which the terminal elements need not all be of the same type. This relational data structure facility is modelled on the concept of associating component members with one of a

number of user defined data equivalence classes. Each component member has an attribute list associated with it and by permitting an attribute to be a pointer to an equivalence class it is possible to build up a dynamic tree to any level.

Data Structures for Algebraic Manipulation

A wide variety of data structures have been used for algebraic manipulation schemes. The operation and characteristics of each scheme are highly dependent upon the form and extent of the data structures used. Many of the data structures have been chosen for a specific application area.

The most primative data structure for algebraic manipulation is perhaps the character string. An expression is stored as a sequence of symbols much the same as it appears in the external representation. Algebraic structure is extracted from the string by scan dominated processes and algebraic operations are performed as transformations and text editing operations. The lack of structure is compensated for by a substantial amount of processing.

Many algebraic schemes have used the data structures of LISP to create and manipulate algebraic data structures. The "cons" function is applied to primative data elements to construct compound structures. These structures may in turn be combined to form more complex structures for subsequent processing through procedures defined in LISP.

ALPAK was the first scheme to involve data structures unique to algebraic manipulation and was written for the IBM 7090. A polynomial is stored as an implicit sum of terms and each term is stored as an ordered set of exponent values. The length of each exponent value is defined by the user and can be up to 36 bits. Any number of exponents can be packed into a word. The term format is also defined by the user

and only terms with the same format can be manipulated. The first word represents the term constant. A polynomial is stored in main store as a pointer, a heading and a data block. The polynomial is referenced through a name table by a pointer. The heading consists of three words the first defining the data address, the second the format address for each term and the third the number of terms in the polynomial. All the terms are stored as a contiguous block. No facilities are provided for sharing common terms or for storing structures on backing store.

FORMAC also has its own data structure for algebraic expressions in which the infix notation of the external algebraic expressions is converted to a special form of Prefix Polish notation. This form of "Delimiter Polish" differs from classical Prefix Polish in that it is not necessary to represent in the data structure all of the operators which are used to define an algebraic expression. In classical Polish notation the string $*\div\div\div ABCDE$ represents $(A\div B\div C\div D)*E$. In Delimiter Polish a sequence of identical operators is replaced by a single instance of the operator but it is now necessary to delimit the scope of this operator. For example if] is used as a delimiter the string $*\div ABCD]E$ again represents $(A\div B\div C\div D)*E$ while $\div A*BCDE]]$ represents $A\div B*C*D*E$. In order to make structural changes in this form of data structure (e.g. say substitute A^2 B for X) it is necessary to recopy the structure with the added changes. Essentially, the structure is always maintained in its canonical format so that operations on any

elements of the structures involve manipulation of the complete structure.

The data structure of SYMBAL is a modified list structure with many of the properties of a codeword scheme. The "knotted list" structure may contain substructures common to other structures. The binary tree type structure of LISP is replaced by an N-ary tree so that variable length elements (as opposed to single words in LISP and double words in SLIP) are involved.

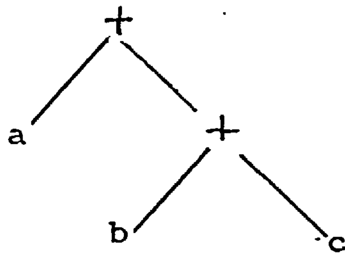


Fig. 3.1 Binary tree in LISP

Each element is essentially a vector of code words which may either directly reference terminal elements or another vector of code words representing a sub expression. In order to reduce the magnitude of the garbage collection problem, all elements are referenced through a common inventory vector. All structure is explicitly defined

through the codeword sequences except for the implicit ordering of code words in a vector. Because of the extensive use of pointers the structure must be contained within addressable storage.

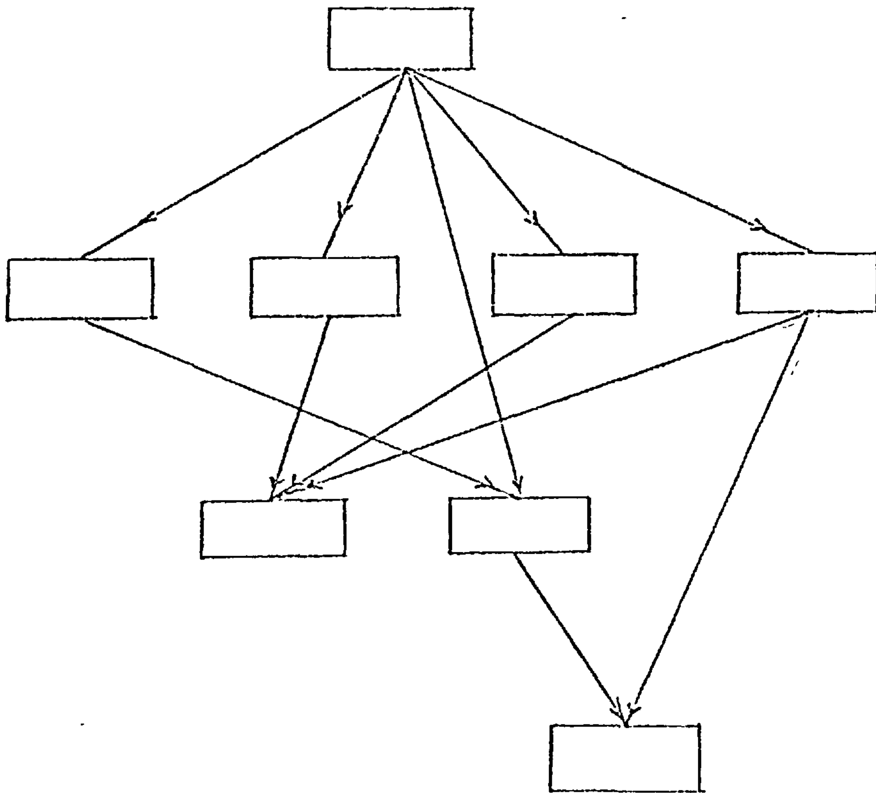


Fig. 3.2 Knotted List Structure of SYMBAL

Summary of Data Structure Requirements

Clearly a comprehensive general data structure scheme suited for algebraic manipulation must provide a number of basic facilities in an economical manner.

The first requirement is that the data structure be truly dynamic in both extent and topology. This can be achieved by either having variable length segments containing the sub-elements or by dynamically combining fixed format segments (e.g. list elements or record class elements) in such a manner as to define the structure. An economy in space for the internal representation will in all likelihood result in a further economy in processing time. It is equally important that accessing of elements in the data structure be performed in a natural and efficient manner.

In order to simplify the subsequent procedures which define the operation on the data structure elements all the algebraic structure implicit in an expression should be explicitly defined in the internal representation. It should be possible to manipulate substructure without involving the complete structure.

Because of the highly dynamic nature of algebraic manipulation garbage collection is often a sensitive, if not critical, area. No data structure scheme for algebraic manipulation can be considered without this in mind and its ultimate consequences.

Most of the algebraic manipulation schemes produced have confined the associated algebraic data structures to to the addressable space available.This often limits the working environment and the size of the problem which can be solved. Where data structures have been relegated to backing store (Ref.21) the overheads incurred have been intolerable. Hence consideration must be given to a canonical format which permits structure and substructures to be readily and economically stored and retrieved from backing store.

Many of the previous objectives may be found to be self defeating for a given implementation as it is unlikely that they can all be achieved in a single data structure, hence some trade-offs may be essential. It is therefore important to be able to offer the facilities which can be moulded by a system designer to suit his requirements. For example the use of common sub expressions can lead to significant reductions in storage space required, however it brings on a host of other problems when garbage collection and the transfer of elements is considered. Where a large work space is available it may not be necessary to be concerned about storing structures on backing store and the storage of common structures may be readily accomodated. AIDS (Algebraic Interpretive Data Structures) has been designed with this in mind, namely to provide a set of basic primative data structure elements and operations suited to algebraic manipulations which can be moulded into algebraic packages with specific characteristics.

Chapter IV

Algebraic Interpretative Data Structure

General Philosophy of AIDS

AIDS is an implementation of a number of basic concepts which provide a general environment for performing algebraic operations. The design objectives include overall system efficiency as well as economy in both processing time and storage requirements. Some restrictions are imposed by the specific implementation although the concepts are independent of any hardware environment.

Fundamental to the AIDS concept is the notion of storing algebraic structure with both symbolic as well as arithmetic data in a single data structure. Each data structure element carries its own "type" with it. The data structure and data structure elements are truly dynamically in both scope and extent.

The facilities provided by AIDS include referencing, accessing and manipulating the algebraic data elements. True recursion is available through the provision of stacks and associated facilities (macros and subroutines) for recursive programming.

The provision of extensive dynamic facilities necessitates the use of sound data management concepts. No single data management scheme is employed, instead

several concepts are used, each selected for its inherent suitability.

AIDS is not designed to be a self-contained system for performing a specific class of algebraic operation. Instead it is intended that it provide a data base from which algebraic compilers for symbol manipulation can be constructed. A number of primitive algebraic functions are provided through subroutines, in much the same way that standard algebraic compilers for numeric computation work by interfacing to associated subroutine libraries in performing standard functions such as square root or logarithmic functions, etc.

The operations of the system defined facilities in AIDS are transparent to the user. The result of algebraic operations will inevitably produce a result which is in a simplified and usually canonical format. Hence further simplification should not be necessary.

It is envisaged that there will be other algebraic operations (e.g. implementation of transform functions, differential equations, etc) that may require considerably more structure manipulation than is performed by the existing AIDS subroutines. The structuring and associated manipulative facilities in AIDS permit this, however in such cases the onus is on the user for some aspects of storage management and simplification. There is always access to the facilities of AIDS for simplification and free store when necessary.

Much of the manipulative power in AIDS arises from the ability to create sets of topologically equivalent structures. The simplest form of a data structure for an algebraic element is a standard canonical format for that element. A structural element may however be built up through an extended addressing mechanism involving both indirect addressing as well as referencing other externally (usually standard or common) defined elements. A further degree of structural complexity can be introduced by having sub-elements which are themselves composite and hence composed of sub-elements etc. This facility is recursive to any degree.

The algebraic operations provided in AIDS do not require that the operand elements be in a reduced or canonical format. However, it is possible to create super-structures (through non-standard type operations) in which conflicts in structure would ultimately have to be resolved. This situation could arise where super structure forces major changes on the structural organisation of component sub-elements[1]. Wherever this situation can potentially arise in the use of AIDS routines it is immediately resolved. In like manner a user departing from the standard facilities must perform the equivalent action. The requirement is not a necessary constraint of the system but rather designed to simplify and expedite the accessing functions.

AIDS is essentially a re-entrant facility requiring each user to have access only to his own data spaces (work spaces, stacks, tables, structures, etc.). Again this is

transparent to the user. No attempt is made to provide common structures to several simultaneous users, however this is not conceptionally difficult to realise.

The present chapter describes the concepts used, as well as the details of the 360 implementation, for creating and manipulating AIDS data structures. The defined algebraic operations are discussed in the following chapters.

[1] For example, such a situation could arise where a composite element of type "term" is affected by an explicit exponent. If a component sub-element (factor) is of "simple" type then the organisation of the simple element must be drastically changed (see discussion of REDUCE function, Page121).

Description of AIDS Data Structures

The data structure chosen for AIDS attempts to meet the demanding requirement for a comprehensive symbol manipulation system. It combines the desirable features of pointer based systems with the freedom of mobility associated with segmentation schemes.

The primitive data elements, as in other symbolic systems, are numbers and coded symbolic values. These primitives are combined to form simple algebraic "typed" data elements. More complex algebraic data structure elements are formed by combining simple algebraic elements in structural relationships. In this manner recursive data structures are defined.

Each data structure element (i.e. excluding primitives) has associated with it control fields for defining algebraic type, referencing and accessing information for sub-elements, as well as fields for defining dynamic extent values. As with other dynamic systems the data elements are manipulated interpretatively. However within any element, referencing and accessing may be either implicit or explicit.

The main function of AIDS is to model and manipulate algebraic formulae. Even though the user's concern is only with the external representation of algebraic formulae, it is informative to understand the internal representation as well as the mechanics of transformation between the two formats. The system is relatively insensitive to user misuse, however there may be extensive system action and

re-organisation which could be eliminated by appropriate user action.

External Representation

The external representation of an algebraic expression is analogous to the representation of algebraic expressions for arithmetic calculation in other languages such as FORTRAN and ALGOL.

Syntax of Algebraic Expression

The complete syntax for the external representation of an algebraic expression is summarised in Appendix I. The syntax is described formally in Backus Normal Form (BNF) terminology as used in the definition of ALGOL 60. Blanks may appear anywhere in an expression as all blanks are removed before the syntax analysis. The following syntactic entities are defined in the external representations using the same referencing system as in the Appendix.

1. Basic symbols

<basic symbol> ::= <letter> | <digit> | <delimiter>

1.1 <letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|
P|Q|R|S|T|U|V|W|X|Y|Z|

1.2 <digit> ::= 0|1|2|3|4|5|6|7|8|9|

1.3 <delimiter> ::= <operator> | <separator> | <bracket>

1.3.1.1 <arithmetic operator> ::= <add operator> |
<multiplication operator> | <exponentiation operator>

1.3.1.1.1. <add operator> ::= + | -

1.3.1.1.2 <multiplication operator> ::= * | /

1.3.1.1.3 <exponentiation operator> ::= **
1.3.2 <separator> ::= , | · | ; | :
1.3.3 <bracket> ::= (|)

2. <u>Identifiers</u>

2.1 <letter digit string> ::= <letter> |
<letter digit string> <letter> |
<letter digit string> <digit> |

2.2 <identifier> ::= <letter digit string>

3. <u>Numbers</u>

<number> ::= <integer> | <rational> | <real>

3.1 <unsigned integer> :: <digit> | <unsigned integer> <digit>

3.2 <integer> ::= <unsigned integer> | <add operator> <unsigned integer>

3.3 <rational> ::= <integer> " / " <integer>

3.4 <real> ::= <integer> " . " | " . " <unsigned integer> | <integer> " . " <unsigned integer>

There are implementation restrictions on the representations for each type of number. All numbers in the data structure are represented as rationals. Hence all reals are translated into rationals and care must be exercised to avoid the loss of significance. If a real constant cannot be properly represented as a rational, then it must be represented by a parameter which can be replaced by a real when the expression is evaluated. The evaluation of all expressions is in single precision real (floating point) arithmetic. This can readily be extended to double precision if the need is warranted.

4. Variables

<variable> ::= <simple variable> | <subscripted variable>

4.1 <variable identifier> ::= <identifier>

4.2 <simple variable> ::= <variable identifier>

4.3 <array identifier> ::= <identifier>

4.4 <subscripted variable> ::= <array identifier> " (" <subscript list> ") "

4.5 <subscript list> ::= <subscript expression> | <subscript list> " , " <subscript expression>

4.6 <subscript expression> ::= <simple algebraic expression>

The variables in algebraic manipulation are not associated with arithmetic values. Each variable name is a representation of itself. For arithmetic evaluation of an algebraic expression, values must be assigned to or associated with the variables.

5. Function Designator

5.1 <function designator> :: <variable identifier> " ("<parameter list>") "

5.2 <parameter list> :: <parameter> | <parameter> " , " <parameter>

5.3 <parameter> ::= <simple algebraic expression>

The function facility permits the representations of standard arithmetic, trignometric and user defined algebraic functions. It is also used to represent algebraic functions such as differentiation or integration.

6. Algebraic Expressions

6.6 <primary> ::= <number> | <variable> |
<function designator> | " (" <simple algebraic expression> ") "

6.7 <factor> ::= <primary> | <factor> " ** <primary>

6.8 <term> ::= <factor> | <term> <multiplication operator> <factor>

6.9 <simple algebraic expression> ::= <term> | <add operator> <term> | <simple algebraic expression>
<add operator> <term>

The syntax for an algebraic expresssion as described above corresponds to the syntax of arithmetic expressions in both ALGOL 60 and FORTRAN IV. The form of this syntax has been maintained to permit interfacing an AIDS type facility to these languages for a hybrid system permitting both symbol manipulation and extensive arithmetic processing.

Internal Representation

An external algebraic expression is modelled by a unique internal representation. The data structure constituting the internal representation contains both algebraic values (constants and variables), as well as topological information pertaining to algebraic structure. Each algebraic expression is represented as a tree with each node defining the structure (type) and number of the next level sub-components.

Algebraic Values

The algebraic values stored for algebraic symbol manipulation in an expression are the names of the variables as well as the numeric values of any constants. In 360 AIDS each variable is represented as being 4 characters in length and hence occupies a full word on the 360 implementation. Longer names are truncated on the right and shorter names are blank filled on the right. The internal representation of each character is EBCDIC, corresponding to 8 bits per byte.

In order to prevent any possible effects due to round off error the internal representation of numbers is binary integer values. However, to allow greater scope for numbers, each number is actually represented as a rational number made up of 2 integer values. Each integer value for the 360 implementation is a half word value giving a range of -32767 to 32767. All integer and real numbers appearing in an external algebraic expression are converted to rational numbers before being stored.

Further Syntactic Entities

The internal representation (data structure) makes use of further syntactic entities to economise on storage and permit quick access to the algebraic values associated with syntactic types of the external representation. The syntactic definitions for these extensions also appear in Appendix I.

The following syntactic groups are used in the data structure:

6.1 <variable exponent pair> ::=<simple variable> " ** " <number>

A variable raised to a constant power is stored as an ordered pair, the variable occupying the first word and the exponent, as a rational number, the second word.

6.2 <simple variable group> ::= <simple variable> | <simple variable group> " * " <simple variable>

A simple variable group constitutes a product of a number of simple variables.

6.3 <variable group> ::=<variable exponent pair> | <variable group> " * " <variable exponent pair>

A variable group constitutes a product of variables one or more of which has an explicit exponent.

6.4 <simple factor group> ::=<simple variable group> | <number> " * " <simple variable group> | <number>

A simple factor group is associated with any combination of a leading number and/or simple variable group. If both number and simple variable group appear they must be separated by an " * ". A simple factor group is represented in storage as a block of consecutive words, the first being a rational number for the constant followed by the algebraic values in the simple variable group, one per word. The order is the same as appears in the external expression except if modified by a subsequent algebraic operation. A control word defining the

length and type is associated with each simple factor group.

6.5 <factor group> :: <variable group> | <number> <multiplication operator> <variable group>

A factor group is the internal representation of a product of variables each raised to an exponent power and an associated leading constant. A control word defines the type and length of a factor group which is again stored as a contiguous block in storage. A leading constant is always stored with either a simple factor group or factor group even though it may be implicit (i.e. = 1) in the external expression. The type field of each associated control word defines the context in which the factor group or simple factor group appears e.g. as a term or a factor. In essence a simple factor group differs from the factor group in that the exponents associated with the variables are implicit (=1) in the simple factor group.

The term AB**2*D*C**3 would be represented as a factor group as follows in storage:

S,I,T	4	8
	1	1
AB		
	2	1
D		
	1	1
C		
	3	1

Figure 4.1 Internal Representation of Term A.B**2*D*C**3

Representation of Algebraic Structural Information

Algebraic structural information is represented through the use of control words. The format of the control word is as follows[1]:

1. algebraic type field - bits 0-3

bits	value	description
0	0	elementary item - either a simple factor group or factor group
	1	composite item composed of sub-elements each of which is either composite or elementary
1	0	implicit exponent(s), (= 1/1)
	1	explicit format for exponent(s)
2-3	00 01 10 11	type term factor simple algebraic expression function designator

2. accessing information field, bits 5-6

bits	value	description
4	0	item stored contiguously
	1	item must be accessed indirectly as specified by Format 2. (Bits 8-31 constitute address of new control word)
5	0	(presence bit) item defined within this structure
	1	externally defined item
6-15	number of proper sub-elements	defines the number of sub-elements associated with the data element
16-31	contiguous length of element <u>or</u> offset to next equivalent element.	defines either the overall length of this element or the offset to the next equivalent element.

1 bit positions numbered left to right 0-31 for 360.

Format 2

The control word is interpreted under Format 2 when the indirect bit is equal to 1.

bits	value	description
0-4		same as Format 1
5	1	indirect addressing
8-31		address of complete new control word.

Through the use of the indirect addressing facility it is possible to create amorphous structures anywhere in main store. There can be any number of levels of indirect addressing. This facility is useful for dynamically altering the structure without undergoing major copying operations.

For elementary items bit 1 defines the exponent format, viz. implicit or explicit. For items composed of a simple factor group bit 1 = 0 while bit 1 is equal to 1 for a factor group .

Representation of Exponents for Composite Elements

For a composite element (bit 0 equal to 1), an explicit format (bit 1 equal to 1) requires storing an extra element as the exponent. This exponent is stored as either an algebraic term or simple algebraic expression data element. The exponent is stored as a separate sub-element after the last sub-element in the element. It is however not considered as a sub-element and hence does not appear in the sub-element count.

Definition	Representation			Value Represented
SAE CW	C,I,SAE	4	31	
Term 1 CW	C,E,SAE	2	9	$(A + B)^2$
Subterm 1 CW	S,I,T	1	3	A
		1	1	
	A			
Subterm 2 CW	S,I,T	1	3	B
		1	1	
	B			
Exponent CW	S,I,T	0	2	2
		2	1	
Term 2 CW	S,I,T	2	4	2A*B
		2	1	
	A			
	B			
Term 3 CW	C,I,T	2	17	$A*(A^2 - B)$
Factor 1 CW	S,I,F	1	3	A
		1	1	
	A			
Factor 2 SAE CW	C,E,SAE	2	13	$(A^2 - B)$**A
Term 1 CW	S,E,T	2	4	A^2
		1	1	
	A			
		2	1	
Term 2 CW	S,I,T	1	3	- B
		- 1	1	
	B			
Exponent CW	S,I,T	1	3	A
		1	1	
	A			
Term 4	S,I,T	1	3	CEF
		1	1	
	C E F			

Fig. 4.2 Canonical Representation of Simple Algebraic Expression
(A + B)** 2 + 2 * A * B + A * (A ** 2 - B) ** A + CEF

For some types of algebraic operations (e.g. differentiation) it would obviously be more convenient to have it the first element as it would involve less retrieval time. However this change could be implemented by making some changes to the system.

Function Designator Representation

A function designator is described and stored in a standard format. The function description is stored as the first sub-element. It is expected that this element will always be a simple item. The number of sub-elements in the count field specifies the number of parameters plus one (for the first sub-element). The definition of the length field remains unchanged.

The parameter elements are stored as sub-elements of the function designator data element. If the function has an explicitly defined exponent this will again appear after the parameter sub-elements.

Generally a simple function name will be stored as a double word simple element; the first word being the control word for the simple element while the second stores the function name.

Definition	Representation			Value Representation
FD CW	C,FD,E	2	3	$SIN^2(X)$
			2	SIN
	SIN			
	S,I,T	1	3	X
		1	1	
	X			
	S,I,T	0	2	2
		2	1	

Figure 4.3 Internal Representation of $SIN^2(X)$

This concept can be extended to encode partial or total derivatives for dealing with differential equations. The "type" of the first sub-element control word is given a value according to its use as follows:

Value	Definition
0	simple function name
1	total derivative
2	partial derivative

The number of variables involved in the differentiations is stored in the count field of the first sub-element. Each variable of differentiation and its order is stored in an analogous manner to the format of a simple explicit element. The length field is as before.

FDI	4	18	name sub element
O2	3	8	
F			
X			
	2	1	
Y			
	3	1	
Z			
	1	1	
STI	1	3	1st parameter
	1	1	
P			
STI	1	3	2nd parameter
	1	1	
Q			
STI	1	3	3rd parameter
	1	1	
R			

Figure 4.4 Internal Representation of $\frac{\partial^6 F(p,q,r)}{\partial x^2 \partial y^3 \partial z}$

Implementation Restrictions

The 360 implementation imposes some size restrictions. For example the maximum length of an element, including simple algebraic expressions, is 32K words (128K bytes). If necessary this could be readily extended by using a double word for the control word. Alternatively a structure could be built up by simple references to other existing structures through the use of the "presence" bit facility. Through this technique each node of the tree is limited to effectively 32K components.

Organisation of Elements within a Structure

The organisation of elements within a data structure can be either implicit or explicit. There may be many equivalent forms of a data structure, each structure differing only in topology.

A structure is composed of elements, each element of which may be made up of other simple or elementary (hence terminal) elements or composite elements which may themselves be further decomposed. The elements of the structure can be dynamic in length at all times. Furthermore, the number of elements in a structure, (or the number of sub-elements in an element), is also dynamic. It is necessary to be able to add new elements at any time as well as deleting other partial or complete elements.

The data structures of AIDS attempt to meet these needs, as well as provide economy in space and accessing time, by combined implicit ordering as found in arrays with explicit addressing of pointer based systems. This form of structure is well suited to a dynamic environment. However it is unsuited to storing structures on backing store. For this the canonical form of the data structure must be used.

Canonical Form of Data Structure

The organisation of elements in the canonical format of the data structure involves only implicit ordering of the elements. This format is used for storing a structure as a contiguous block or segment, either in core or on

secondary storage. Figure 4.2 represents an expression stored in canonical format.

The map for determining the organisation and the topology of any data structure is embedded in the control word sequence of the structure. Each control word associated with an element defines the make-up of the element. For simple elements the relevant topological information is the length of the element.

Composite items are made up of consecutive sub-elements stored in contiguous storage locations. The control word for a composite element defines both the number of sub-elements and the overall length of the element. Each control word contains sufficient information for passing from one control word to the next equivalent control word. For sub-elements the control word of the first sub-element is found as the first word past the element control word. Addressing in the canonical format is essentially through relative base and offset.

Extended Form of Data Structure

The canonical form requires that the data structure be stored in consecutive locations in storage. Any modifications or changes will usually involve rewriting a new structure, unless the new elements are of the same length. In order to readily permit modifications to an existing structure without extensive rewriting, a pointer based facility is available through the use of an indirect addressing mechanism.

The format of an indirect address control word is given by Format 2. If the indirect addressing bit is equal to 1, then the address of an updated control word is found at the address specified by bits 8-31 of the current control word. This itself may involve indirect addressing etc. to any level, although more than two levels are not likely to be required.

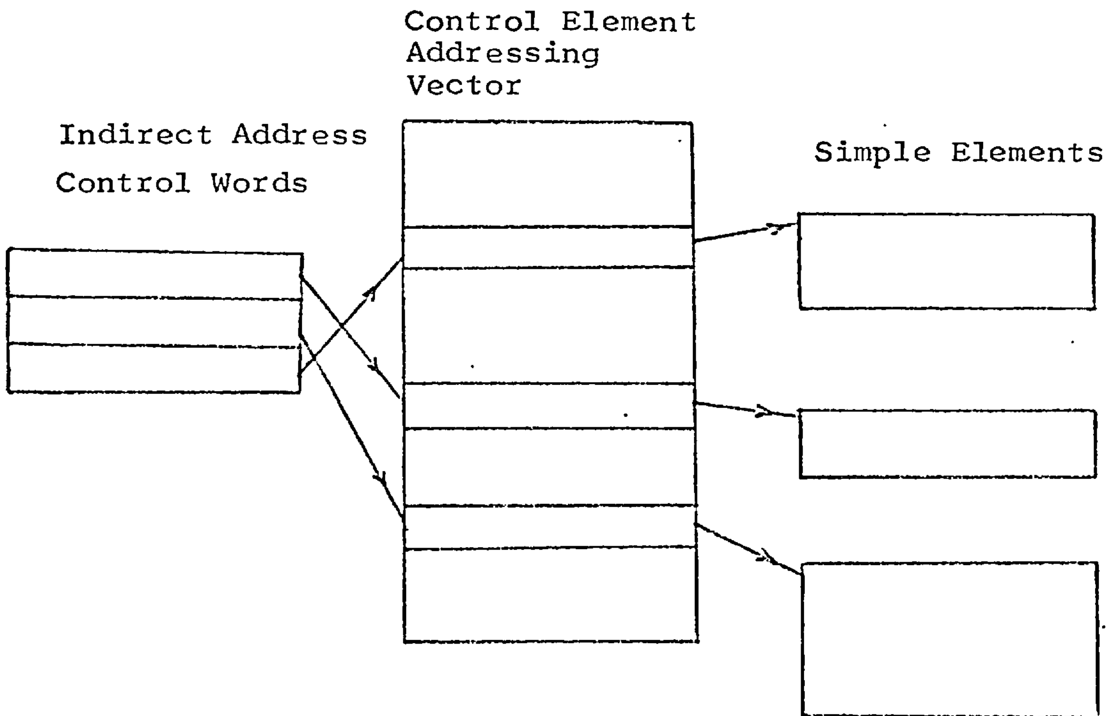


Fig. 4.5 Addressing Algebraic Data Elements through Common Element Table.

An instance where this facility proves useful is to permit accessing common elements without duplication. In the canonical format an algebraic expression is essentially stored as it appears in the external representation. There is no facility for avoiding redundant representation except through the use of the presence bit for identifying common external simple algebraic expressions. However when working in a dynamic environment where common forms (i.e.

algebraic items) are likely, each common form could be addressed through a common element table. This will permit changing all instances of the item by making only the one single change in the element itself. However, it is for the same reason that a change in a single instance would induce changes in all instances that non redundant representations are extremely dangerous and of limited value.

The indirect addressing format defines the addressing for only a single element, which can be either simple or composite. The next equivalent element after an indirect addressed element is referenced implicitly through the control word immediately following the previous indirect control word (see figure 4.6). This element itself might be indirect, etc.

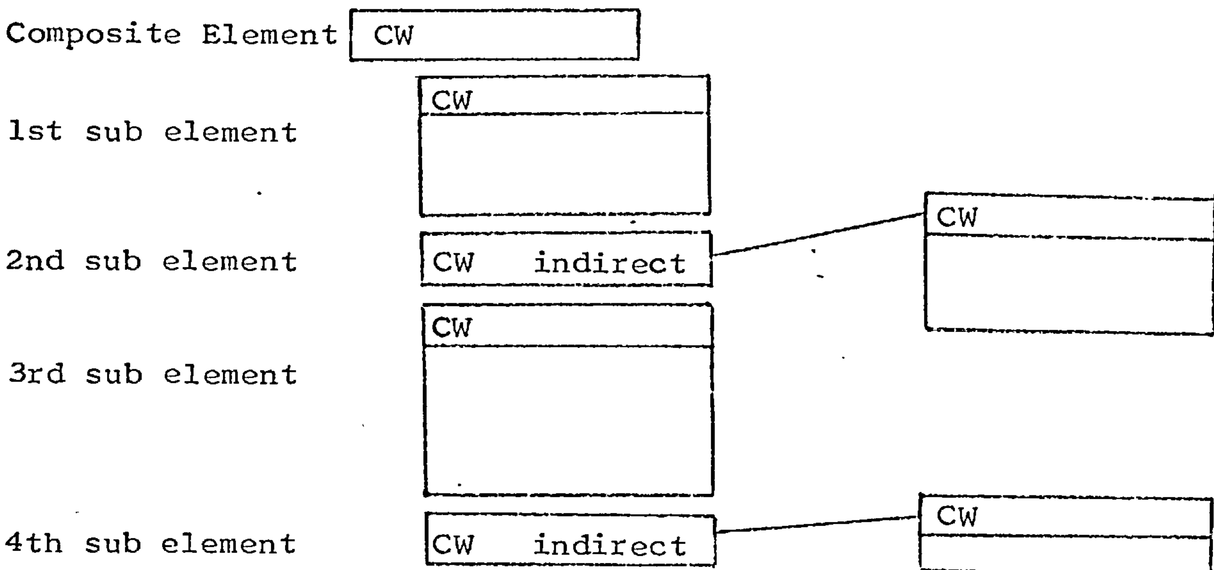
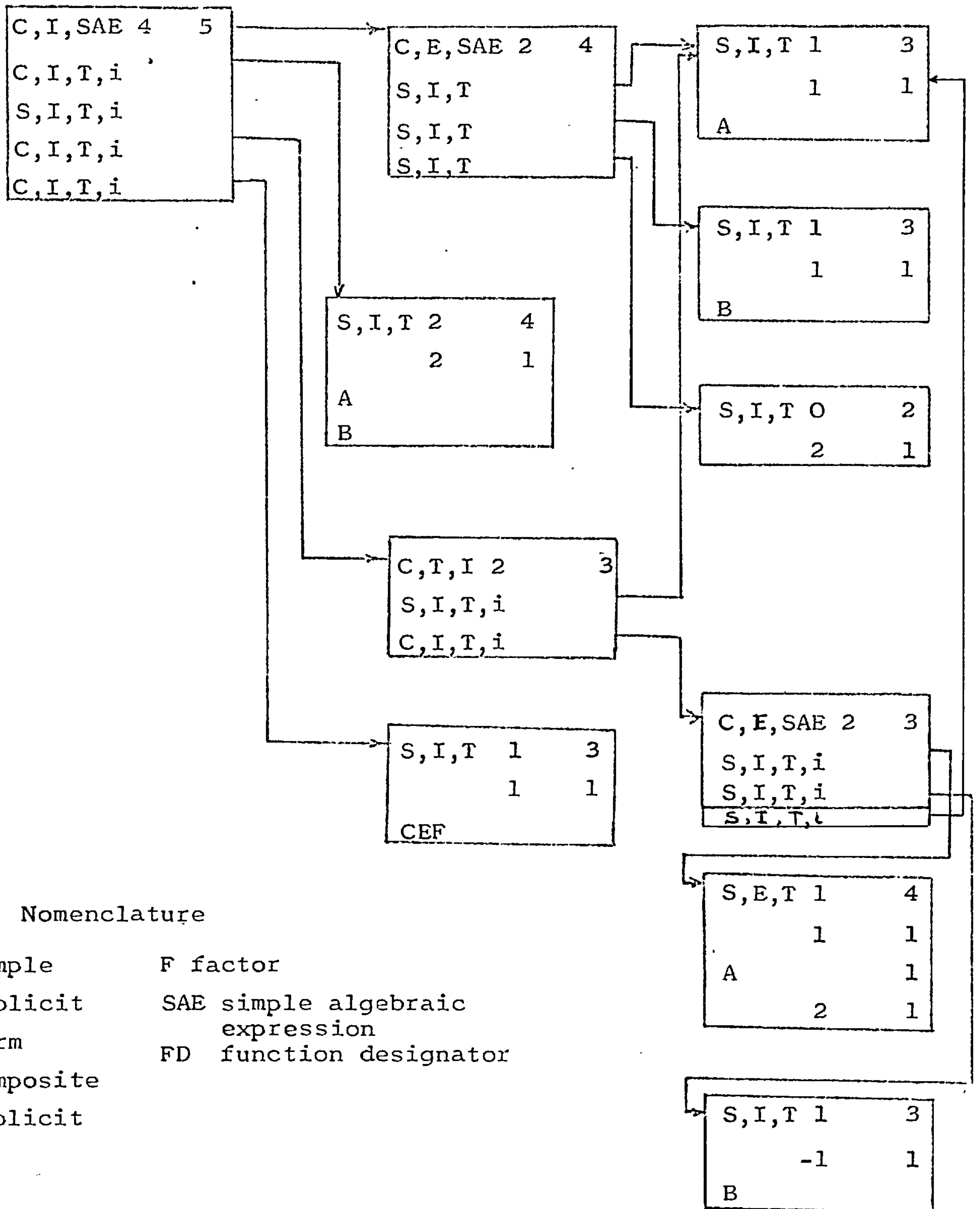


Figure 4.6 Mixed Format Composite Data Element



Nomenclature

S simple
I implicit
T term
C composite
E explicit

F factor
SAE simple algebraic expression
FD function designator

Fig. 4.7 Representation of Simple Algebraic Expression
(A+B) ** 2+2 * A * B+A * (A ** 2 - B) ** A+CEF
in Extended Format

Of course it is possible to build up the data structure by having vectors of control words at each level and pointer to the final terminal elements. Figure 4.7 shows this type of data structure for the expression (A+B)**2+2*A*B÷A*(A**2-B)**A+CEF.

The control word mechanism provides very effective methods of manipulating algebraic structure without involving excessive data movements. In this way it is possible to build up complex algebraic elements from existing simple elements. For example, if simple data elements exist for $3 * A^2 * B$ and C these can be combined to form a simple algebraic expression by defining an element of 3 control words as in Figure 4.8.

The first control word specifies that this element is a simple algebraic expression with implicit exponent consisting of two terms. The total length of contiguous words is specified as 3. The second and third words of the element are each of type term and each is flagged as specifying the address of the control word where the item may be found (i.e. Format 2 control word).

This element could in turn be referenced from another data element at a higher level. Figure 4.9 illustrates a situation where the element defined in Figure 4.8 is used as a factor by a higher level element. In this manner a hierarchy of algebraic structure can be created from simple data elements.

Whenever structures are created in this manner,

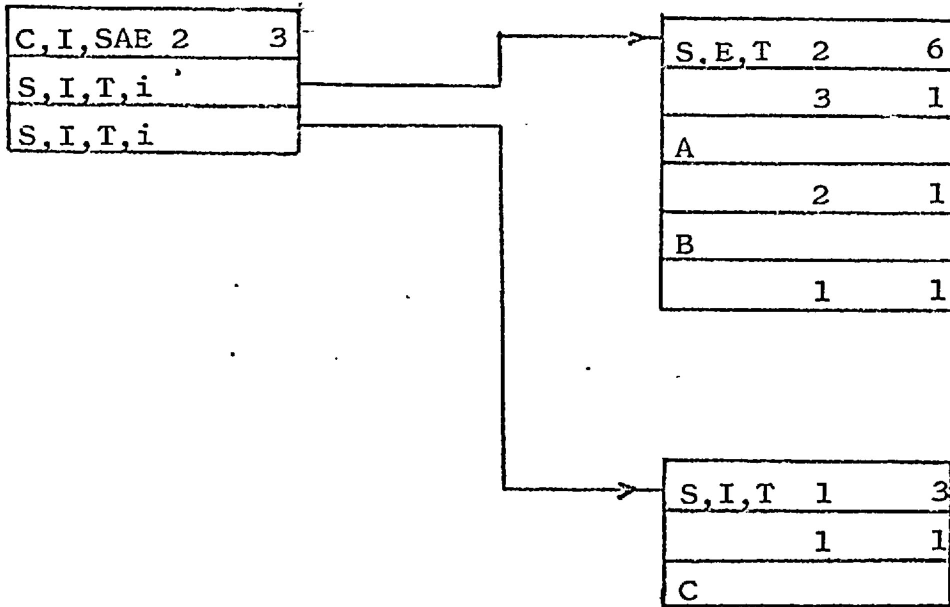


Fig. 4.8 Representation of Simple Algebraic Expression $3A^2B + C$

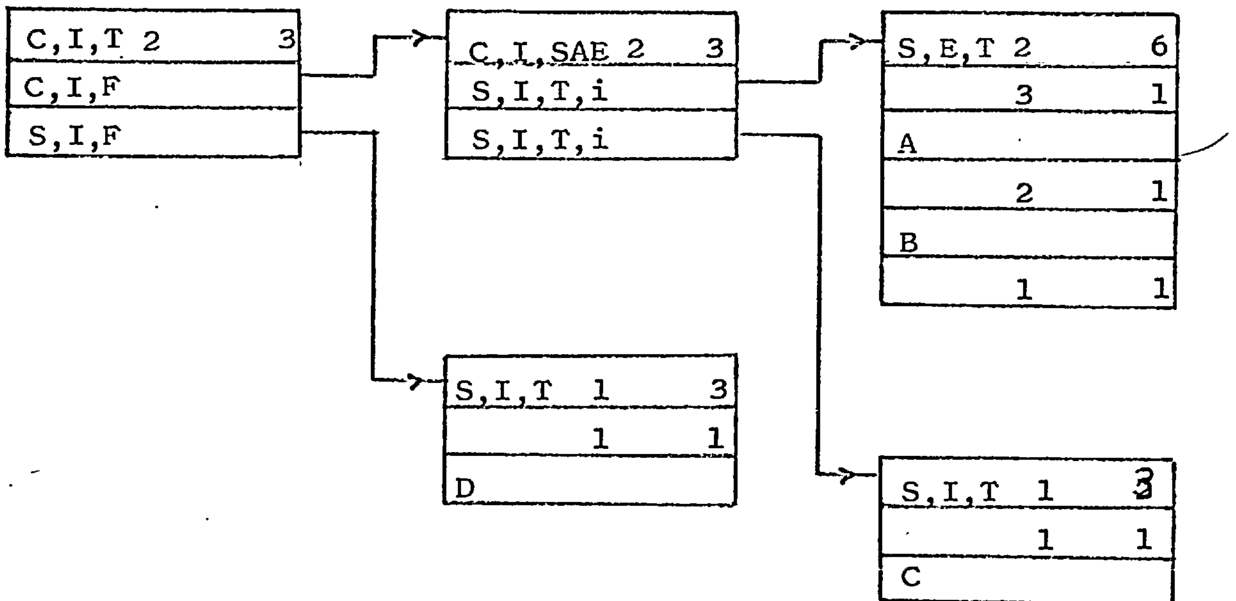


Fig. 4.9 Representation of Term $(3A^2B + C)D$

the following rules are defined for subsequent referencing and accessing:

1. Whenever an element is referenced through a control word where indirect addressing is specified, algebraic structure is built up through the control word sequence.

2. The organisation of a terminal data element is ultimately determined from the control word of the terminal element. Hence simple or composite type can only be deduced from the terminal element itself. The algebraic type, however, is specified by the higher level control word (e.g. the data element is to be used as a term or factor etc.). In this manner the algebraic type of simple data elements can be overridden.

3. If a control word specifies both indirect addressing and an explicit exponent, the exponent control word is specified in the word following the indirect address control word. In this way it is possible to affect a complete algebraic structure by an explicit exponent.[1] If the terminal element itself is specified as having an explicit exponent, the exponent element is found in the normal manner.

[1]This is a non standard type of operation which produces structures whose components cannot be properly accessed through the structure macros. The element can be reduced to a standard format by the REDUCE routine.

Further Extensions

The presence bit is a facility for referencing externally defined elements which are themselves simple algebraic expressions. When the presence bit equals 1, bits 8-31 of the current control word define an offset into an external expression table for subsequent reference to the appropriate simple algebraic expression.

Accessing Elements within the Data Structure

The data structure which models any external algebraic expression is basically a tree of any number of levels, with any number of branches at each node. No element in the structure can be directly accessed without tracing through the tree, except of course if previous known pointers to specific elements have been kept. Most operations in simple algebra involve treating only one level down at any one time, although of course this facility can be recursive. For example, multiplying 2 expressions involves access to each expression as well as to the terms in each expression, which are at the next level down. Normally one is not concerned with locating a specific element, but rather with passing through the structure performing specific operations on all or parts of the structure.[1]

A number of macros have been implemented for facilitating access to the elements in a structure. These macros only permit accessing elements at the same level or the next level down. This facility can be used recursively at any level in the structure.

1. NEXTD RP=2, RD= 3

The NEXTD macro is a keyword parameter macro for locating the first sub-element (i.e. the leftmost

[1]In the next stage of development this will not be strictly true as it is hoped to use a visual display for user interaction. Again however it is only necessary to associate a position on the display with an algebraic entity. Even though complex items can be represented on the display the actual trace time will still be negligible.

son) of any element. The keyword parameter identifies the register containing the address of the current element control word. The address of the first sub-element control word is returned to the register identified by the keyword parameter RD. (In accordance with the rules for keyword parameters, if no macro parameters are specified default values of registers 2 and 3 respectively are used.)

2. NEXTE RP=2, RE=3

NEXTE is also a keyword macro, used for locating the address of the control word for the next equivalent (i.e. right brother in the tree) element. The parameters RP and RE are similar to those in the NEXTD macro.

3. ELEMENT PR=2, ER=3

Through the use of the indirect addressing facility the body of an element could be found only at the end of a chain. The ELEMENT macro effectively produces the final address of the control word for an element. The keyword parameter PR identifies the register storing the initial control word address while ER identifies the register to which the final address will be returned. The ELEMENT macro will continue processing until a terminal condition is reached.

4. EXPONENT RP= 2, RE= 3

The EXPONENT keyword parameter macro locates the address of the control word defining the exponent data element. The address of the composite data element whose address is sought is contained in the register specified by the keyword parameter RP. The exponent address is returned in the register specified by the keyword parameter RE.

5. FUNCTION RP= 2, RF= 3

The FUNCTION keyword parameter macro is used to locate the address of the first element of a function designator data element which defines the function. The keyword parameter RP identifies the register containing the address of the function data element. RF specifies the register to which the element address is to be returned.

Conversions Between External and Internal Representations

An external algebraic expression is read in as a character string and must be converted to the internal representation. For output it is necessary to be able to convert from the internal data structure to a formated line of EBCDIC characters. The first conversion requires extracting the algebraic values as well as the algebraic structure for subsequent storage in the data structure. The latter conversion is performed by a special purpose print routine.

The data structure corresponding to an algebraic expression is built up during a syntax analysis of the expression. The recursion facility within AIDS is used for the creation of the recursive data structures.

Operation of the Syntax Analyzer

The function of the syntax analyzer is to convert an algebraic expression from the external syntactic representation (see Appendix I) to the internal syntactic representation.

The syntax analyzer itself operates in a conventional reoursive manner by using 2 local stacks, each of which stores single word values. The first stack stores the return address to which control is returned after a search for a syntactic item. The second stack stores the address of the string pointer on entry to a syntactic

entity recogniser routine.

In constructing the syntax analyzer advantage has been taken of unique hardware instructions in the 360 which assist considerably in syntactic analysis, namely the TRT (translate and test) and TM (test under mask) instructions. The TRT instruction works in conjunction with a 256 byte function string which is used as a byte table. The TRT instruction has two operands, one of which points to a variable length source string (or substring) being analysed while the other operand identifies the function byte string. Each byte in the source string is used as an offset to reference a byte in the function byte string. If the function byte has value zero the operation is repeated with the next byte in the source string until a non zero function byte value is found or the length of the source string is exhausted. If the source string is exhausted without producing a non zero function byte, this condition is signalled by setting the condition code equal to zero. When a non zero function byte is produced the operation ceases and returns the absolute machine address of the source byte in register 1 along with the value of the function byte in register 2. By this means it is possible to set up a function byte string to search for delimiting characters. However, it is more useful as a means of classifying each byte type and the TRT instruction is used in this manner in AIDS.

The function byte string (of length 256 bytes) is created in AIDS by the STRTABLE macro using the following keyword parameters:

&NS	non-valid symbol in source string	non graphic symbol
&D	digit	0-9
<	upper case letters	A-Z
&AO	add operator	+ -
&MO	multiplication operator	* /
&RO	relational operator	> = <
&LO	logical operator	& | ¬
&S	separator	, ; . : .
&DL	delimiter	( ) ' "
&SS	special symbol	? @ £ % _
&BLK	blank	

It is relatively easy to change the table as well as incorporate new class entries such as a lower case letter region. The STRTABLE macro also creates single byte values for each class entry. Consequently there is a single byte for the categories DIGIT, LETTER, ADDOP, MULTOP, RELOP, LOGOP, SEP, DELIM, SPEC, NONSYM and BLANK set with the value of the corresponding keyword parameter. The name assigned to the table produced by STRTABLE is named CODESTR and is included within the code for the syntax analyzer. This is not included in the user area as it remains invariant with processing.

To assist in writing and modifying the syntax analyzer a small number of useful macros have been defined:

1. BOT TYPE, ADDRESS

   The macro BOT (branch on type) can be issued after a TRT to effect a branch to the address specified by &ADDRESS if the current character being examined in the source string is of type specified by the parameter &TYPE (i.e. a classification such as DIGIT, LETTER, etc.).

2. BNT TYPE, ADDRESS

   The BNT macro (branch not type) is similar to BOT except that the branch is effected only if the type of the current differs from the type specified by the &TYPE parameter.

3. NEXTCHAR

   The NEXTCHAR macro moves the string pointer up one position.

Syntax Analyzer Conventions

The conventions which have been adopted in the AIDS syntax analyzer do not restrict its capability. Some of the recognizer routines in the syntax analyzer are also used by the command interpreter for recognising syntactic elements by adhering to the conventions.

The syntax analyzer works by examining single characters in turn from the source string. The address of the current character being looked at is always held in register 3. Registers 1 and 2 are always available for use by the TRT instruction.

Each recognizer routine operates without storing current status information such as register contents etc. It leaves undisturbed current status registers and uses the accepted scratch registers for local processing. When creating structures from the source string current status data structure information is maintained in the user's area.

Before entering a recognizer routine it is necessary to stack the address to which control is returned upon completion of the recognizer routine, as well as the current address (pointer) in the source string, in case the test fails. The stacks for storing these values are labelled ASTACK and PSTACK respectively and each permits a maximum of 300 entries (words). Values are stacked by transfering control to a local routine STAKAP which assumes that the contents of register 2 contain the return address and register 3 points to the current source string character to be examined. Each recognizer routine ter-

minates by branching to either the USTAKA or the USTAKAP routine. USTAKA removes the last entry from the stack and branches to the return address without resetting the source string pointer. This constitutes the "item recognised" condition and is signalled to the calling program by setting the length of the recognized item in bytes in register 0. USTAKAP corresponds to the "item not recognised" condition so that register 0 is set to zero and the string pointer reset to its value before entry to the recognizer routine.

An algebraic variable is always kept as 4 characters and hence fits into a single word. Whenever a variable identifier for simple variable, array variable, or function designator is recognised, the corresponding recogniser routine returns the value in register 8. Note that truncation or blank fill on the right will have occurred if necessary. In the same way, all arithmetic values are returned in the appropriate form (integer, real or rational) in register 9. For example, when recognising < variable exponent pair > the variable is returned in register 8 and the exponent as a rational number in register 9.

All routines in the syntax analyzer maintain the integrity of the base registers (10 and 11) for the syntax analyzer as well as the base registers pointing to the user data area.

Forming Algebraic Structures

An algebraic data structure is built up during the syntax analysis as each algebraic data item is recognised. Simple data elements are created in a large work area, while a control word sequence is built up in a separate vector. If the syntax analysis is successful the structure is copied into its own allocated area and stored in canonical format.

Upon entry into the <simple factor group> and <factor group> recognizer routines a block is automatically created in the work area. If the test for the syntactic entity fails, the pointers into the work area are reset to their previous values. If the test succeeds the pointers are updated and the address of the block is stored in an indirect address type control word by a higher level algebraic recogniser routine.

Each algebraic type element has its own local data space in the user data area. As the syntactic elements <simple algebraic expression>, <term> and <function designator> are potentially recursive, the current data values must be saved on a data stack whenever any of the above are re-entered and reset when the previous environment is again invoked.

Conversions Between Data Structure and External Representations

Because of the dynamic and recursive feature of the data structure the transformation between internal and external data representation implies considerably more processing than for more conventional structures. However, as the conversions between the two representations constitute only a small part of any algebraic system, this overhead can be tolerated.

The conversion is performed by print routines which operate recursively. The routine which decodes and prints simple algebraic expressions pass control to a similar routine which prints out each term in turn. Both of these are obviously recursive. All of the necessary editing characters are inserted as required.

Recursive Facility in AIDS

The AIDS system makes extensive use of recursion so that powerful system recursive facilities are required. Unfortunately the 360 hardware is not particularly suited for stack type operations hence these must be effected through software. It is of course possible to perform recursive programming through the GETMAIN and RETMAIN macros in the control program services of OS 360, however this procedure is neither efficient in time nor in its use of storage. As the stack in AIDS can grow very large during processing it is often necessary to save stack copies on secondary storage devices, such as the disc. Therefore, in AIDS, recursive programming is done through a stack which is saved on disc whenever it is necessary. The GETMAIN-RETMAIN facility requires considerably more processing in that the stack segments must be explicitly chained together.

On many systems recursive programming is facilitated through the use of special hardware. (e.g. Burroughs 5500/6500, KDF9, PDP10). On others (Univac 1108) it is more readily possible than with 360 to simulate stack operations through the use of existing registers.

The recursive facilities provided in the present system are coded as macros for stack operations. The system may employ any number of stacks up to a maximum of 256. It is essential that there be at least 2, one of which (STACKO) is used primarily for recursive programming and stacks current register values, while the other is a data stack required by the syntax analyzer when creating a structure. Each stack is described by a set of control values which are stored in the user's data area. A new stack cannot be added dynamically, as with the Burroughs systems, but must however be configured into the system by setting up a new block of associated control values in the user or data area. The control values describing each stack are as follows:

CSSA - current stack segment address

NASSA - next stack segment address

NSS - number of stack segments

LSA - lower stack address

LS - length of stack

USA - upper stack address

TSR14 - temporary storage for register 14

Description of Stack

A stack is made up of variable length stack segments which are each described by a stack segment control word which is the first word of the stack segment. CSSA holds the address of the current stack segment word. NASSA

is the next free word past the present stack segment. LSA and USA are the addresses of the lower and upper stack boundaries, while LS is the length of the stack in words. TSR14 is used as a temporary store for register 14 by the stack macros as explained later. NSS is the number of stack segments currently active.

The stack segment word has 3 fields associated with it, namely:

NR	- bits 0-3	number of registers saved in this segment
LSS	- bits 4-15	length of last segment in words
LCS	- bits 16-31	length of current segment in bytes

NR is stored so as to facilitate the accessing of data when stored in a stack segment along with the register contents. The LSS value permits moving back to the previous stack segment control word when unstacking is required. This value is given in words to permit a maximum size segment of 2^{12} words or 16K bytes. LCS gives the length of the current stack segment in bytes. Byte, instead of word values, are used as offsets for addressing must be in bytes even though each stack is essentially a word stack.

Even though the 360 is essentially a byte machine, the stack and stack operations are based on words.

Recursive Programming Macros

1. RSAVE

A recursive routine is essentially a routine which can call itself. Hence a first requirement for such a routine is that the save areas associated with the calls to the routine be separated from each other. STACKO is used in the system for saving the register contents.

RSAVE is the recursive save macro for saving the contents of the general registers and is described by the model statement:

RSAVE &R1=0, &R2=14, &NR=X'OF', &STACK=0

All of the parameters in this macro are keyword parameters. The parameters &R1 and &R2 specify the starting and ending registers that must be saved. If these parameters are omitted values of 0 and 14 respectively are assumed. The parameter &NR is the number of registers saved and is expressed in hexadecimal form. A default value of 15 is assumed if the parameter is not specified. The &STACK parameter references the stack to be used and has a default to STACKO if no value is specified.

The maximum number of registers that can be saved is 15 as it is not anticipated that saving 16 is necessary. It has been necessary to deviate from 360 conventions to a small degree, however system reliability or performance will not be affected. Registers 14 and 15 are used in the conventional manner. Register 13 no longer points to the current save area as the save area is implicitly

2. RRETURN

The recursive return macro RRETURN is the complement of RSAVE[1]. It operates in essentially an analogous manner by producing in-line coding for restoring the register contents from a stack segment and providing the linkage to a routine USTACK which updates stack status information.

The USTACK routine, much like RSTACK, loads the current stack information, updates and performs checks on it. The number of stack segments is decremented by 1 and the current and next stack segment address words are updated. A check is made to ensure that the stack area is not empty, else it is necessary to reload from disc the previous stack area.

3. STACKSEG and POPSEG

For recursive program not only is it necessary to save current status information (usually reflected in the general register) but also current data values. All such data is usually stored in consecutive locations in core and hence can be saved as a segment along with the contents of the general register. The STACKSEG macro moves a data segment from a user area to the current stack segment. The model statement is:

```
&NAME    STACKSEG    &ADDRESS, &LENGTH, &STACKO
```

The positional parameter &ADDRESS and &LENGTH specifying the data segment address and the length in words

[1] The model statement for RRETURN is:

```
        RRETURN     &R1= 0, &R2= 14, &STACK= 0
```

defined by current stack parameter values, however R13 should be loaded to point to a save area which can be used by the control program or a non recursive routine if needed.

RSAVE will usually be the entry point to the recursive routine and will usually (although not necessarily) be accessed through R15. However R15 is used by the RSAVE macro as a general purpose register for addressing purposes and hence is a volatile register, so that it should never hold a value to be saved. This restriction does not interfere with 360 operations.

The RSAVE macro produces a limited amount of in-line coding for addressing purposes and also saves the registers specified. Part of the in-line coding is the linkage to the routine RSTACK for updating the stack values.

The RSTACK routine performs many of the functions that would normally be handled by appropriate hardware. It loads the data for the designated stack into the general registers, updates and performs checks on it. The number of stack segments NSS is incremented by 1 and a stack segment control word created for the stack segment holding the current values of the registers as stored by the RSAVE macro. A check is made to ensure that the stack is not full (i.e. within 15 words of the upper stack address) and when full the stack contents are written to a disk.

respectively. &STACK is a keyword parameter identifying the stack to be used. If this parameter is not specified STACKO is assumed.

The complement of STACKSEG is the POPSEG macro which moves the data part of the current stack segment to the user area specified in the macro. The model statement for POPSEG is:

&NAME POPSEG &ADDRESS, &LENGTH, &STACK=0

Both STACKSEG and POPSEG produce in-line coding for storing parameter values and linkage to the routine STACKSEG and POPSEG respectively.

Any number of data areas may be added to the current stack segment although it is not expected that the facility will be used in this manner. However, as no information is stored with each data segment to differentiate it from the other data areas in the current stack segment, the unstacking operations work on only one data area. This need not be a limitation of the system as it is possible to add extra coding to unstack data areas from the top of the current stack segment by changing the current length. As it is not envisaged that this enhancement would be particularly useful, the extra overhead to be incurred can hardly be warranted, even though there is more generality and flexibility.

Data Stacks and Data Stack Operations

The main stack (and most used stack) is STACKO which is intended primarily for recursive programming. This allows essentially the stacking of recursive save areas as well as associated data areas if necessary. However, a single stack will not suffice in the system if efficiency is to be an essential requirement. Some operations, such as creating an algebraic structure, require the stacking of data values only. Hence provision is made in the system for any number of data stacks.

The two main macros used for data stack operation are STACK and POP. STACK stores a data area as a stack segment while POP unstacks a stack segment and moves the data to the user area.

The model statement for STACK is:

```
&NAME      STACK      &ADDRESS, &LENGTH, &STACK= 0
```

The keyword parameters &ADDRESS and &LENGTH identify the user data area and length in words respectively. &STACK refers to the stack being referenced and defaults to stack 1 if no value is specified.

Similarly the model statement for POP is:

```
&NAME      POP      &ADDRESS, &LENGTH, &STACK= 0
```

The parameters have the same meanings as in STACK.

The general stack macro STACKSEG and POPSEG can also be used to add and remove a data area to an existing data segment.

The recursive facility is sufficiently general to allow the system programmer to perform complex recursive operations on any number of predefined stacks. Each stack added requires the system to be re-assembled. The dummy control section which describes the user's data area (USER) must be changed to include another set of parameter values which describe the new stack. Some of the values which are referenced through the recursive macros in other control sections must also be declared as entry points.

Data Management

The highly dynamic nature of algebraic manipulation necessitates the use of specialised store management techniques for several aspects of the AIDS system. The AIDS concept is designed to cope with multiple concurrent users in an environment in which all users share the program code in common. Each user has however access only to his own data areas.

The main areas of data and storage management are as follows:

1. a fixed length data area associated with each active user
2. a free storage scheme for the allocation and de-allocation of blocks of any length for dynamic data handling
3. secondary storage facilities for storing structures when not needed immediately in main store as well as for storing overlay data segments
4. cataloging structure facilities for name and variable value tables

User Data Area

Each active user is allocated a data area for storing local data values required for the proper operation of the AIDS routines. All save areas and parameter list data areas are referenced within the users data area[1].

[1]For the 360 implementation, register 12 is reserved for storing the pointer to the user area at all times. The symbolic data references are defined by the dummy control section labelled USER.

Also included in the user area are the general purpose stacks and stack environment values as well as the local stacks of the syntax analyzer. The dynamic data areas within the user area are checked to ensure that they do not overrun their allocation with disastrous consequences. In some cases the same save areas or parameter lists are used by more than one routine but only when the routines cannot interact in any way with each other.

One segment of the user data area, which is not necessarily contiguous with the main data area, is a large scratch work area used by the syntax analyzer and other special routines. This area is not logically essential to AIDS, however by providing a relatively large scratch area it eliminates the need for an excessive number of calls on the free storage facility and for associated housekeeping operations. Again this area is only used by a single process at any one time and is therefore managed by a processing routine. It is used primarily by the syntax analyzer and the expansion routines, essentially in much the same manner as a stack, to create temporary data elements during a process. These elements are moved during a final pass to a structure area.

Stacks

The stacks are associated with the recursive mechanism available in AIDS. Each user area has

stack control values for at least 3 separate stacks. Variable length stack segments described by a stack segment control work are stored and retrieved from the stack on a first in first out basis. When a stack is full this copy of the stack is stored on disk and the stack area overlayed with the new stack extension. Similarly when a stack area is empty and more stack segments exist, the previous incarnation of the stack is reloaded and new status information set. For this the first 4 words of a stack are reserved for holding:

1. the identification (i.e. record number) of the last stack segment so it can be retrieved when needed
2. length of the previous stack segment in words
3. current stack segment address (CSSA) to be used when the stack is re-incarnated
4. next available stack segment address (NASSA) for the next re-incarnation.

Free Storage Scheme

The free storage scheme allocates data blocks (in words) of any size from a pool of free areas upon demand. The size of the free storage area is set at system generation time and will usually involve claiming all the remaining space available in a partition or remaining in main store. There is no set maximum value and it is expected that the system will work satisfactorily with about 32K words in the free store area.

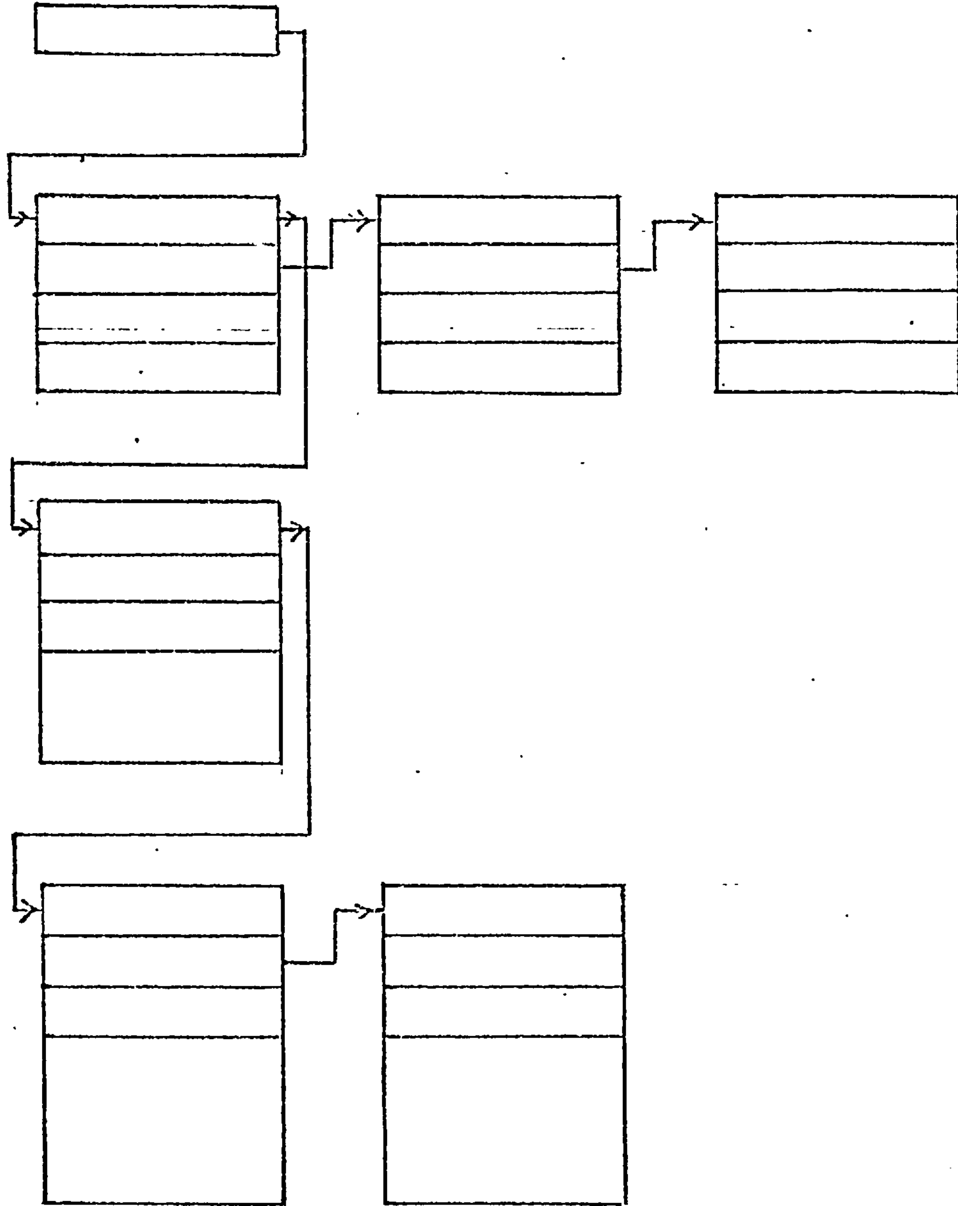


Fig. 4.10 Free Storage list structure

The free storage scheme operates through a two dimensional list structure. All the blocks of different length are chained together, in order of increasing size, in a single chain. All blocks of the same size are chained together in another list. Fig 4.10 represents a typical free list structure.

The first 3 words of each block in the free list are used for storing status and pointer information. Word 1 points to the next largest block in the free list, if one exists, while word 2 points to the next block of the same size, also if it exists. Word 3 is divided into 2 fields, the first byte representing the number of remaining blocks of the same size while the 3 low order bytes contain the length in words of this block.

A request for a block of given length is made to the GETCOR routine. If a block of this size does not exist the next largest block which does not leave a residue greater than a pre specified size is broken up and used. The residue is of course entered in the appropriate position in the free list. This method tends to conserve large areas which are obviously required for holding large structures. If a request cannot be met a condition code is returned to the calling routine[1].

When a block is returned to the free store an attempt is made to attach it to either end of an existing block in the free list. If this can be done the new block is moved to its appropriate position in the free list.

[1]Register 15 returns a non zero value on the 360 implementation.

The free store scheme does not maintain an inventory of blocks allocated and their length but relies on the integrity of the routines which demand and return free store blocks. However as all the free blocks are allocated from a single area of main store, RETCOR checks to ensure that a returned block is within bounds.

Because of the dynamic nature of most manipulative routines in AIDS the length of block required is not always known a priori. Hence the usual procedure is to request a larger block than is needed and return the unused space.

<u>Secondary storage facilities</u>

Because of the nature of algebraic operations, the structures can grow very large even when in a reduced or simplified form. It is essential that a back up facility, for storing structures not immediately required, be provided. Also it is required to be able to save stack copies so that the stacks are not limited to a fixed length. Two routines are provided in AIDS for storing and retrieving variable length (in words) records; DSAVE for saving and DGET for retrieving records.

DSAVE accepts as parameters the address from which the record is to be written and its length in words. After the record is stored DSAVE returns a record identification number, for subsequent retrieval

to the calling routine. DGET is the complement to DSAVE and uses the same parameter sequence. The calling routine must provide both an identification number as well as a length and address. The length value could be maintained by the secondary storage mechanism. The length value for structures is kept in the structure name table for structures and in the stack extension area for stack areas.

Catalogue Facilities

AIDS provides and maintains a catalogue of all named structures for any user. A separate name-value table is maintained for associating a value for variables used in the structures. The format of the structure and name-value tables is given in Appendix V. A number of routines for maintaining the catalogue are provided.

The routines INSERTN inserts a name and value into the variable name table. INSERTS is an analogous routine for inserting a name into the structure name table and a pointer to its location in main store. If entries already exist they are overwritten. DELETEN and DELETES delete variable name and structure entries respectively from their associated tables. ADDSTR returns the address of a catalogued structure and NVALUE returns the current value associated with a variable.

Chapter V

Algebraic Operations

A number of system defined operations are available for manipulating the elements of the data base in AIDS. Arithmetic operations on numeric representations consititute the degenerate case for formal algebraic operations. As with arithmetic operations on numeric data the most primitive operations are embedded in the instruction repetoire. More complex operations are defined in terms of the primitive operations and constitute essentially built-in function subprograms. A user defined function procedure represents yet a higher level of functional complexity.

Most conventional digital computers have been designed almost exclusively for numeric data processing. Hence the instruction repetoires do not usually incorporate complex instructions, except of course for character handling, designed primarily to assist in non-numeric processing. However as the trend is to build special purpose hardware for specific operations such as the IBM array processor (refs 33), combined with the emergence of complex numeric operators such as found in APL (ref 34), it is to be expected that the complexity

of operations performed by the hardware will continue to increase. In an analogous manner it should be possible to define algebraic operations that could readily be realised by micro programmed hardware sequences.

The essential difference between operations on numeric entities and operations on symbolic entities is that the extent of the result of the latter is dynamic while numeric calculations produce well defined results. The result of a symbolic process often requires a second pass to reduce it to its minimal form in terms of a canonical representation.

Some languages permit single operations on complete data structures or on their multi-element components. PL/1 for example, has structure operations which essentially constitute an implicit DO loop for a single operation on all elements of the structure or sub structure. The operation must of course be valid on the primative elements, although conversions may be induced. In PL/1 the structures are inherently static and the operators can be any of the arithmetic, comparison or string operators. These facilities are not readily adaptable to symbolic processing in the AIDS context. Useful operations for symbolic processing can be found in some high level languages such as SNOBOL (ref. 24) For example the balanced string variable is useful for isolating a balanced pair of

parentheses. Also, APL defines vector type operations by single operators.

The algebraic operations performed in AIDS can be divided into several distinct classes. The operands are the data elements as defined in AIDS, which can be of either simple or composite type with either implicit or explicit exponent formats. The following classes of algebraic operations are defined in AIDS.

1. arithmetic
2. logical
3. expansion (removal of parentheses) and factoring
4. replacement (substitution)
5. simplification
6. conversions
7. differentiation and integration
8. evaluation
9. functions
10. data movement

Simplification, being such an important operation in any symbolic system in that it provides an index of merit for any such system, is devoted to Chapter VI. Substitution and removal of parentheses and factoring, being related operations, are also discussed in Chapter VI. Differentiation and integration are essentially procedures based on the operations 1, 2 and 4 and are

dealt with in Chapter VII. Conversions between internal and external representations were described in Chapter IV. The remaining operations, 1 - 3 and 8 - 10 are also discussed in this chapter.

Many of the operations involve manipulating the algebraic data elements. Often however these operations can be performed by manipulating only the control words and making a subsequent pass on the data elements to produce a result in canonical and simplified format.

The operations in AIDS are available as standard procedures. Some of the operations are recursive while most require access to the user's area. All of the routines are written as re-entrant procedures.

Data Transmission

The input-output of variable length records (which can of course be structures) was described under Data Management in Chapter IV. Over and above this facility is the requirement to be able to move a complete data element from one part of the main store to another. A single data element in AIDS may represent a very complex structure with many sub-elements in a truly recursive manner. Further, a data element may be either in an extended format, with many indirect addressing links, or in canonical format. There is no way of knowing whether a complex structure is stored in a canonical format from the high level control words. Only the terminal elements need be in canonical format.

A single recursive routine is provided for moving a data element referenced by its control word from one store location to another. This routine, MOVELEM, has as parameters the address of the control word for the element to be moved and the address to which it is to be moved. The resultant element is stored in canonical format with updated control words. No simplification is performed except to remove any indirect addressing links.

MOVELEM can be used to move simple elements. However, if the element to be moved is known to be

simple then it is possible to perform a move operation in-line by using the MOVE macro. The model statement for the move macro is as follows:

MOVE LR=5, FROMR=6, TØR=7

Each parameter references a register; LR specifies the register which holds the length of the block to be moved in bytes, FROMR specifies the register containing the address of the element to be moved and TOR specifies the resultant address. If no parameters are specified the default values for LR, FROMR and TOR are 5, 6 and 7 respectively. Register LR goes to zero after the operation and FROMR and TOR registers are updated to point to the next available byte.

Arithmetic Operations

Arithmetic operations are defined on the data elements of AIDS. The algebraic type field of the simple or composite operands must be either " term " or " simple algebraic expression ". The arithmetic operations which are defined include addition, subtraction, multiplication and division for both simple and composite operands. No routines exist for performing exponentiation although this can be done by substitution. Exponentiation is treated as a special case rather than a basic primative operation in algebraic manipulation as any element in AIDS can be affected by an exponent through the use of the control words. A simplification mechanism can, in this case, be invoked to reduce the construction to canonical format. Arithmetic operations on simple elements involve both arithmetic and symbolic data. The degenerate case in which there is only numeric data associated with the simple data elements involves only rational arithmetic.

Rational Arithmetic

All numeric primative data values (excluding fields in the control words) are stored as rational numbers. For the 360 implementation the numerator

is stored in the most significant half of the word and the denominator in the remainder. A number of routines in AIDS provide a rational arithmetic capability.

Rational addition is performed by PRADD which adds two rationals to form a rational result[1]. PRSUB is an analogous routine, also with 3 parameters, for rational arithmetic subtraction. In like manner PRMULT and PRDIV provide rational multiplication and division respectfully. All result values are reduced to simplest terms by application of Euclids greatest common divisor algorithm. If the result value generated is too large an error message is printed. In its present form the rational arithmetic package can be rather explosive. Consideration should perhaps be given to dealing with rational numbers as ordered word pairs in store, in much the same manner as complex values are maintained in most systems.

[1] Rational addition on 2 rationals a/b and c/d is defined as $\frac{ad+bc}{bd}$

Symbolic Addition and Subtraction

Symbolic addition and subtraction are defined on data elements which are of type "term" or "simple algebraic expression". Addition and subtraction are dyadic operators requiring two operands to produce a resultant operand. Hence all the corresponding routines have as parameters, the addresses of the two operands and the resultant except for the routine ADDTERM. This latter routine attaches a term to an existing simple algebraic expression by adding an indirect address term control word to the expression control word sequence.

The routine ATERMS performs symbolic addition on two terms. The result may be either a single term or a simple algebraic expression consisting of two terms. If either one of the term operands is a simple algebraic expression the result will consist of the second term operand being added to the expression. If both operands are of type "simple algebraic expression" the result will be a single simple algebraic expression involving the terms of both expressions. If it is desired to maintain the parentheses of a simple algebraic expression used as a term, ADDTERM should be used. In the case where both terms are of type "term" a test is made to see if the two terms can be combined in whichcase the result is a single term. A result is always in simplified format.

The routine ASAES adds two simple algebraic expressions to form a resultant simple algebraic expression. This routine operates by forming a vector of indirect address term control words. A simplification mechanism[1] is then applied to move and reduce the result.

Symbolic subtraction is performed in a similar manner, making provision for sign of the second operand, by the routines STERMS and SSAES respectively.

Simplification is implicit in all of the above-named routines except ADDTERM.

Multiplication and Division

Multiplication and division involve more variety than addition and subtraction. When both operands are simple elements the result is formed by essentially concatenating the body (data elements less control word and constant) of the two elements together as well as updating the resultant control word and producing a new constant. Both simple elements must be of the same exponent type else an expansion from

[1] HLC routine - see Chapter VI

implicit to explicit exponents will be required. The result is always reduced to its simplest form by the COMPRESS routine (see Chapter VI)

Division of one simple element by another is somewhat more complex. The dividend is stored in the result area with explicit exponents (this may of course involve expansion from implicit to explicit format). If the divisor has implicit exponents the divisor body, with explicit exponents of -1/1, is concatenated with the dividend. The resultant control word and constant are appropriately modified. If the divisor has explicit exponents, all exponents are changed in sign as they are moved. In either case the result is simplified as with term multiplication.

The simple data element operands for the above operations can be either of type "term " or " factor " and the resultant data element will have the type code of the first operand. It is expected that the type code will be overset by the calling routine in cases where ambiguity could arise.

Multiplication and division where either or both operands are composite elements is performed by creating an intermediate element consisting of a control word sequence. The composite elements may be either simple algebraic expressions or composite terms, and each is handled by separate routines. A

composite factor is treated as a composite term.

The routine TMULT performs multiplication of two composite terms (or factors) to produce a simplified resultant term. The operation is performed, rather simply, by creating in the user's data area an intermediate composite term with implicit exponents consisting of two indirectly addressed control words (of type simple term with implicit exponents) pointing to each operand. Control is then passed to the term simplification routine RTERM (see Chapter VI) for simplifying the intermediate term and moving the result to the location specified by the result operand.

TDIV is an analogous routine for performing term (or factor) division. It operates in exactly the same manner except that the second control word is set to type " simple term with explicit exponent ". The exponent comprising a simple element with value -1/1 is stored after the second control word. However before the result can be formed it is necessary to pass control to the routine (REDUCE) so that the exponent -1/1 can be applied to all subelements of the division. REDUCE produces a control word sequence which is passed to RTERM in order to form the resultant term element.

One of the operands in either TDIV or TMULT

may be a simple element. No expansion is performed by either of these routines. If a component factor of a term is a simple algebraic expression it will also appear as such in the resultant term. For example in multiplying the composite term $3A^2B^3(A + B)$ by the composite term $A(A + B)^2$, the resultant term will be $3A^3B^3(A + B)^3$. Division of the first by the second will correspondingly produce a resultant term of $3AB^3(A + B)^{-1}$.

Multiplication and division involving two simple algebraic expressions are performed by the routines MSAES and DSAES respectively. Again the operations are performed by manipulating control word sequences followed by a subsequent simplification process. MSAES operates by representing, in a work space, the first simple algebraic expression as a sequence of indirectly addressed control words, one for each term. A second sequence of control words is created to represent the resultant simple algebraic expression. Each element in this sequence is a composite term consisting of two factors. The first factor references a term in the second multiplication operand while the second factor references a term from the previously created control word sequence. All possible cross products terms are created and then the resultant simple algebraic expression is simplified by the HLC routine (see Chapter VI).

Division of simple algebraic expressions is done in a similar manner. Each term of the divisor divides each term of the dividend. The same type of control word sequence is created as in MSAES except that the second factor is modified by an explicit exponent of -1/1 before simplification.

Logical Functions

The logical functions which can be applied to the algebraic data elements involve testing for both algebraic structural equivalence as well as equivalence of symbolic values. Two routines are provided for this purpose, one of which pertains to tests on simple elements while the other is only applicable to composite algebraic elements. Each routine returns a condition status byte (see Appendix IV)

Equivalence of Simple Elements

The routine TESTESI tests two simple operands for equivalence. The simple operands can be either of implicit or explicit format. No test is made on the algebraic type field (e.g. factor or term) as this information is readily available to the calling program when needed through the appropriate control words.

The first test performed is on exponents. If both operands are implicit then this test is skipped. If both operands are explicit a test for equivalence of variable exponent pairs is performed. However, if one operand is of implicit format when the other has explicit exponents then the explicit exponents must either all be plus ones or minus ones. If the exponent values of one operand are the inverse of those in the other, this

condition is reflected in the return condition byte. This facility can be used to facilitate the search for common factor groups in rational function expressions[1].

As there is no system -imposed ordering of the variables according to a collating sequence, a search for equivalence of variables can be very lengthy. However, it is unlikely that a search will proceed to any depth if the simple operands actually do differ. As no search is made if the number of variables is not identical in each operand, the only condition that leads to excessive search time is when both operands have many common variables, all of which are concentrated at the beginning of the first operand.

If the variables and exponents match a further test for equivalence of constants is made.

<u>Equivalence of Composite Elements</u>

Tests for equivalence on composite elements are performed by the routine TESTECI. As a composite element represents a tree or subtree, this routine is recursive. Essentially it is similar in structure to TESTESI and also returns a condition byte. It is necessary to identify a mismatch and default as quickly as possible because of its potentially time consuming nature.

[1] It is proposed to extend the existing routine to search for a subpattern within a given operand.

The first test is to check the number of elements in each composite element operand. If they differ there is an immediate default. Care must be exercised with operands in extended format to ensure that the algebraic structure as reflected in the control words is the same for both operands[1]. This particular test is searching for topological equivalence.

If the previous test is successful each subelement in the first operand is matched in turn against each subelement in the second operand. Failure to match on any scan produces an immediate default. When both subelements being matched are simple TESTESI is called; when both are composite a recursive call to TESTECI is made, else the scan continues.

[1] For example a term with n factors would not be considered equivalent to another term of n - 1 factors where one of the factors is made up of 2 subfactors even though the 2 terms are symbolically equivalent.

Functions

A function facility is required within algebraic expressions for providing a higher level of generalisation of the algebraic data elements. This facility can be used for defining in-line representations for a number of elements. For example, a term in an expression may represent a Bessel function of given order etc. The same function facility is used to represent total and partial derivative as explained in Chapter IV.

The function facility can aslo be used to define a selector function for a data element (or elements) on a group of existing data elements. It may for example, be appropriate to define a selector function for identifying a term data element from within a simple algebraic expression which has the highest power in a given variable.

The algebraic data elements within a function description (e.g. < simple algebraic expression > , <term > etc) are treated in the same way as the data elements within a simple algebraic expression. The creation and maintenance of these elements is left to the special function processing routines provided by the user.

Numeric Evaluation

It is often necessary to find the numeric value of a symbolic expression. This can either be done by substituting numeric values for symbolic variables wherever possible and simplifying as is done in FORMAC or associating a value with each symbolic variable in an evaluation procedure.

EVALSAE is the AIDS routine which evaluates a simple algebraic expression and returns a numeric result. This in turn calls EVALTERM for evaluating a term which may in turn call EVALFD for evaluating a function designator. All routines are recursive and all calculations are performed in single precision floating point arithmetic.

The routines operate by associating numeric values with the symbolic variable. The value of each variable is found from the variable name table. If a variable name cannot be found in this table, the structure name table is searched and if found this structure is immediately evaluated for a numeric result. Failure to identify a variable causes a message to be printed and the evaluation process terminates.

Removal of Parentheses

Removal of parentheses is a linearising operation which reduces algebraic structural relationships within a simple algebraic expression to a minimum. This process involves both multiplying out terms and expanding expressions raised to an integer power. Both operations are part of the simplification process in FORMAC under user control. These operations are performed in AIDS by manipulating control word sequences in a recursive manner.

Removal of parentheses is performed on a term by term basis within a simple algebraic expression by the routine EXPANDS. It requires as parameters the address of the simple algebraic expression and the address where the result is to be stored.

Each term is formed in a scratch area by removing one level of parentheses at a time. The resulting simple terms are removed to a resultant area while further expansion is introduced if parentheses still exist. The simple terms are simplified before being moved. The resultant expression is subsequently simplified.

The expansion of a simple algebraic expression raised to an integer power is performed by creating a vector of term control words in a large scratch area. Mulfiplication is performed using only indirect address control words until the exponent power has been reached. Each term is then

simplified followed by a simplification of the resultant expression.

Removal of parentheses is undoubtably the most complex operation performed by AIDS.

Factoring

No attempt has been made to implement the inverse operation to expansion, namely factoring. However, the facilities for implementing factoring are available for user defined procedures. Further facilities are desirable such as searching for a subpattern within a simple element as well as extracting the largest common pattern from two or more simple elements. Neither of these is difficult to implement. It is clear that factoring should be attempted in a highly interactive environment, such as with visual display terminals, where extensive user direction and control is possible.

Chapter VI

Algebraic Simplification and Substitution

Simplification

Central to any sophisticated symbolic algebraic system is the need for an efficient simplification mechanism. The simplification is usually related to the canonical format of symbolic representations within the system. Simplification is essential in that it removes redundant symbolic expressions as well as permitting more efficient processing on the resultant data structure elements.

A brief survey of the development of simplification routines is given in reference 10. The authors cite several independently written routines for performing some aspects of simplification. Most of these are LISP based except notably their own simplification subsystem within the FORMAC system called AUTOSIM.

The exact meaning of simplification is not readily definable and is usually only to be found in terms of a working philosophy for a given algebraic system. For example most would agree that $(a + 3a + 4b)$ should be reduced to a representation of at least $(4a + 4b)$ and preferably $4(a + b)$. However, it cannot always be expected that the term $(a - b)(a + b)$

should be automatically reduced to $(a^2 - b^2)$ as it may be more informative to the user, expecially in an interactive system, to maintain the original representation. The removal of redundant values is invariably desired, however, the transformation of an algebraic entity to another form should only be performed under a set of well defined conditions, all of which can be controlled by the user.

The importance of simplification cannot be overstressed as simplification of a resultant operand can in many cases account for more processing time that the algebraic operations which produced the resultant element. The essentials of simplification are a pattern matching operation combined with the creation of an equivalent representation for an algebraic element. Again, because this is a dynamic type of operation, demands will be made upon the data management facilities in the system. Simplification in LISP based systems (refs 10, 11) consists usually of a number of recursively defined routines which operate on their list data structures. The arguments against this type of facility are essentially those against all list processing systems, namely inefficiency in processing time and storage requirements.

The AUTOSIM package in FORMAC differs from previous attempts at simplification in many pronounced ways.

It is a complete subsystem encompassing many aspects of simplification as defined within FORMAC. AUTOSIM does not employ recursion in the same manner as LISP based systems. It does however make use of a push down store. Fundamental to the simplification process in AUTOSIM is the ability to flag, and subsequently test for) elements which are in simplified format.

FORMAC's authors have taken the view that a substitution of a variable for an expression should be performed whenever possible. For example consider the following FORMAC statement:

1. LET E = (A + B) **N

2. LET F = (A + B) **M - E^2 + 5 * E ** D

The first FORMAC statement defines an expression E to have the value (A + B) ** N. If when this is encountered at object time B is an atomic variable (e.g. symbolic) and A and N are FORTRAN variables with values 1 and 2 respectively then the representation for the first expression will then be (1 + B) ** 2. If subsequently expression 2 is encountered and M is a FORTRAN variable with value 6, and D an atomic variable, the representation for this expression will be (1 + B) ** 6 - (1 + B) ** 4 + 5 * (1+B) **2*D. If however further simplification is possible at this level it will be performed. For example if D were to have a FORTRAN value of 3, the resulting representation for

expression F would be

$$6 * (1 + B) ** 6 - (1 + B) ** 4$$

The algebraic operations performed in FORMAC rely on the representation of expressions to be in a simplified canonical format.

AUTOSIM performs the following "natural" simplification transformations:

1. $0 ** A \rightarrow 0$ where $A \neq 0$
2. $1 ** A \rightarrow 1$
3. $A ** 0 \rightarrow 1$
4. $A ** 1 \rightarrow A$
5. $(-A) ** N \rightarrow \begin{cases} -A ** N & \text{if N is an odd integer} \\ A ** N & \text{if N is an even integer} \end{cases}$
6. $-(-A) \rightarrow A$
7. $EXP (LOG (A)) \rightarrow A$
8. $LOG (EXP (A)) \rightarrow A$
9. $-(3 * A * (-B) * C * (-D)) \rightarrow (-3) * A * B * C * D$
10. $\sum_{j=1}^{n} A_j \rightarrow \sum_{\substack{j=1 \\ j \neq k}}^{n} A_j$ where $A_k = 0$

    $\prod_{j=1}^{m} B_j \rightarrow \prod_{\substack{j=1 \\ j \neq k}}^{m} B_j$ where $B_k = 1$
11. $\prod_{j=1}^{m} B_j \rightarrow 0$ where there exists at least one value of k such that $B_k = 0$

Most of the above transformations are usually inherent in any simplification system. However, AUTOSIM applies further transformations under control of the user. These relate to the evaluation of standard functions with numeric parameters. The options governing these transformations are :

a) evaluate all functions automatically

b) evaluate only the integer-valued functions (factorial and combinatorial)

c) evaluate only the transcendental functions (EXP, LOG, SIN, COS, ATAN, TANH)

d) no functions to be evaluated .

AUTOSIM is a scan driven process and operates by deciding whether a simplification transformation is applicable to the part of the expression currently being scanned. Each algebraic operator governs the simplification operation on its associated operands and the lower level operators. The applicability of the simplification transformations can be determined from a transfer table specifying the association between operators. Part of the decision process is to perform contextual checking. There are essentially three main types of context which may be checked before applying a simplification transformation. Firstly there is a check for specific patterns of operands and operators (e.g. the simplification of B ** (-K) is only done if K is an integer). Secondly a test is made to check if the

sub expression has already been simplified. Lastly, transformations are only applied after checking mode switches for specifying simplification options.

All transformations are not immediately applied when first recognised in the transfer table but are delayed until sub expressions are simplified. In this way the need to perform some transformations may be eliminated. Also intermediate in-line growth of expressions can be reduced (e.g. the transformation $(T_1 * T_2 * \ldots. T_n) ** X \rightarrow (T_1 ** X \ldots.. T_n ** X)$ should only be performed after the base $T_1 * T_2 \ldots..T_n$ has been simplified) by delaying simplification at one level. Simplification in AUTOSIM involves an ordering of all operand variables for an operator in delimitor Polish notation.

Essentially FORMAC simplification (AUTOSIM) is a scan dominated process for applying transformations on the internal representation of an expression. The process involves a complex set of rules for both determining when and how a simplification transformation should be applied. Extensive data movements with sorting and merging are involved.

Simplification in SYMBAL achieves essentially the same objectives in a somewhat more formal manner. The simplification mechanism consits of a series of procedures which may be called recursively to reduce and modify the

list data structure for an expression. A transfer table is used to determine sign whenever parentheses are removed. SYMBAL also uses five modes which determine the level and extent of any simplification. These modes provide controls for:

1. distributive multiplication: It is possible to specify the removal of all parentheses up to an integer power. Parentheses will be retained for all expressions having larger integer exponent values.

2. delayed assignments: The user may control the assignment of a value to a variable through this mode

3. common denominator: This mode makes it possible to control whether rational expressions are to be represented as

$$\frac{\sum(\text{numerator terms})}{\text{denominator}}$$

or as $$\sum(\text{numerator terms/denominator})$$

4. truncation of power series: For power series representations all terms with an exponent greater than a specified value are dropped.

5. distributive multiplication in expressions consisting of a single term: For expressions consisting of a single term, distributed multiplication is unconditionally suppressed.

Individual routines exist for simplifying factors and terms. Most make extensive use of a scratch area and ultimately result in copying the simplified element into a new area.

Simplification in AIDS

Algebraic simplification in AIDS is performed either implicitly as an integral part of an algebraic system operation or explicitly through a user initiated command. Simplification is performed in accord with the needs of proper system operation.

In much the same manner as with other extensive algebraic systems the algebraic data elements are always maintained in reduced canonical format. Operations which produce results which may require simplification automatically invoke the necessary simplification mechanisms. Some of the same simplification mechanisms can also be invoked through user control.

Since simplification is such a vital part of any algebraic manipulation scheme it is essential to be able to perform this function on any algebraic data element as economically as possible in terms of processing time and storage space required. The result produced by any basic simplification mechanism should be a data element in canonical format with no redundant representations. The redundancy argument applies to both structural as well as symbolic data.

One of the basic reasons for choosing some of the data structure elements found in AIDS was to make the simplification process as natural and simple as possible. Hence a minimal amount of manipulation is performed in

a well defined manner.

Simplification of elements in AIDS usually involves re-writing the simplified element into a location different from that of the original element. This approach has been chosen primarily because of expediency. There are cases where a simplified element in AIDS may require more storage space than the unsimplified element. In this case there would be an unjustifiable amount of housekeeping and data movement involved. The result of any simplification in AIDS is to produce a simplified element in reduced canonical format. This is not a necessary condition of AIDS operation and an analogous simplification mechanism could be constructed to produce simplified elements in the extended format. This facility might perhaps be useful for a highly interactive environment such as the use of visual displays where obviously it would be undesirable to rewrite large elements because of only minor changes.

The various levels of simplification in AIDS can be broadly classified as follows:

1. simplification of simple algebraic elements
2. simplification of composite algebraic elements
3. reducing structural complexity to canonical format.

Simplification of Simple Algebraic Data Elements

The simple algebraic data structure elements in

AIDS consist essentially of contiguous lists in store containing control information as well as algebraic primatives according to a format defined by the control word. A simple element is dynamic in length and hence can contain any number of symbolic variables or variable exponent pairs, but only one element constant. Simplification of a simple element involves removing redundant variables from the element. For example the unsimplified simple element 3 * A ** 2 * B ** 1 * A ** 1 would simplify into 3 * A ** 3 * B ** 1.

The simplification of simple elements is performed by the COMPRESS routine. This routine accepts as parameters the address of the simple element to be simplified as well as the address of the location where the simplified element is to be stored. COMPRESS operates by selecting each variable in turn and scanning for its next occurence, if any, in the element. If it does not occur again the variable (with or without exponent) is recopied into the new area. If, however, another instance of the variable is found the exponent is accordingly updated and the scan stopped. In this case the scan for the next variable begins without recopying. This process can be executed in the same data area in which the term exists only if the simple term has explicit exponents. For the case where the simple element has implicit exponents a match during the

scan necessitates expanding the simple item from an implicit exponent format to an explicit exponent format before the previously described process can be applied. (This is essentially the reason for rewriting simplified elements into new areas of store - i.e. an element can grow in length during a simplification operation!)[1] This expansion is performed in the result area and COMPRESS is then applied to this element.

Whenever a match of variable names is found the exponents are updated by rational arithmetic addition. If the result is zero the variable exponent pair is not included in the new element being formed. The control word for the resultant elenent is automatically updated during the operation.

The EXPAND routine accepts as paramenters both the address of a simple element to be expanded from implicit exponent to explicit exponent format and the address of the location in store where the result is to be formed. This operation involves merely inserting exponent values of 1/1 for all variables.

The COMPRESS, in conjunction with EXPAND routine, performs the following simplifications in AIDS:

[1] The break-even point occurs when the number of variables in the original element is 2 (equating lengths gives $2 + N = 2 + 2(N - 1)$, hence $N = 2$).

a) A ** 0 → 1 (implicitedly)

b) A ** M * A ** N → A ** (M + N)

c) $\prod_{i=1}^{m} F_i \rightarrow \prod_{i=1}^{n} F_i$ where m - n (m > n) variables (F_i) are common

The scans in the simplification process could be shortened by ordering the variables within a simple element according to the collating sequence in much the same manner as AUTOSIM. However, AIDS attempts to avoid this restriction so that a user defined order may be maintained whereever possible for possible use in highly interactive systems. In any case, the amount of processing time is not likely to be reduced significantly as the ordering process could itself consume considerable processing time.

Simplification of Composite Algebraic Data Elements

Composite elements in AIDS combine algebraic structural information with symbolic data of simple algebraic data elements. Simplification of composite elements involves both structural re-organisation as well as combining and simplifying simple elements.

The composite syntactic elements < term > and < simple algebraic expression > are simplified by the routines RTERM and HLC respectively. Each routine accepts as parameters the address of the composite

element to be simplified and the address in which the result is to be stored.

Term Simplification

Term simplification involves reducing a term to the minimum number of factors by combining redundant representations. The RTERM (reduce term) routine makes two passes on the given term, first to remove all simple factors and combine them into a single simple factor element and then a second pass to extract all composite factors. The resulting simple factor resulting from the first pass is itself reduced by the COMPRESS routine. If the resultant constant is zero the resultant control word is set to a single word of zeroes signifying a simple term of length zero (i.e. a null term). If the constant has value 1/1 and further composite factors exist, the simple item is not removed, although it could well be. The overhead borne by maintaining a simple factor of value 1/1 is small and it can prove useful when performing further algebraic operations.

The second pass involves comparing each composite factor with all remaining composite factors in the term for either complete or partial equivalence. If the two factors differ only in exponents a resultant factor with updated exponents is moved to the resultant

area. This scan cycle continues for further possible matches of composite factors for the same partial equivalence. Each composite factor as it is used, is flagged so that it can be ignored in subsequent scans. New scan cycles are initiated until no further composite factors remain.

The effect of RTERM is to produce a reduced composite term in canonical format. A simple factor, if one exists, will appear as the first sub element. All composite factors with common base will be combined into a single composite element of the same base but updated exponent. In essence RTERM performs the following simplification:

d) $0 * A \rightarrow 0$

e) $\prod_{i=1}^{m} C_i \rightarrow \prod_{i=1}^{n} C_i$ where m - n $(m > n)$ composite factor have base elements in common with other elements

The control word which describes the new term is updated to reflect any changes as a result of the simplification.

Simplification of Simple Algebraic Expressions

The simplification routine which reduces the representation of a simple algebraic expression is HLC (high level compresser). This routine operates in much the same manner as RTERM. Each term in turn is

compared against all remaining terms for either complete or partial equivalence, the partial equivalence in this case being "differ in leading constant value only". Matches which lead to possible reductions involve storing an updated term in the resultant area as well as flagging the term in a separate map area to signify that it has been accounted for. The order of the terms is not changed and where reduction has occured the position of the first appearance of the term is maintained .

HLC assumes that each term is in a reduced state from the operation which formed it. The control word is updated to reflect any changes induced by the simplification mechanism. The result of combining two terms may lead to a zero result which is checked. Simplification from HLC is defined by

f) $C * A \rightarrow 0$ where $C = 0$, and C is the constant associated with a simple data element

g) $\sum_{i=1}^{m} T_i = \sum_{i=1}^{n} T_i$ where m - n (m > n) terms differ from other existing terms in no more than a constant factor

Resolving Structural Complexity

Complex algebraic structures can be built up by describing and referencing sequences of control words. By this means it is possible to build up superstructure

on existing algebraic and symbolic data. The most important instance of this is the case where all factors in a term or all factors in a composite factor can be affected by an exponent. Some or all of the factors may already have explicit exponents and the term or factor may have one or more (if the composite item is not in simplified format) simple factors.

The routine REDUCE has been designed to transform this type of structural complexity for a term or composite factor into a standard format. REDUCE accepts as parameters the address of the element as well as the address where the result is to be stored. The effect of this routine is to produce a vector of control words and data elements in which each base element is identified by an indirect address control word followed by the complete exponent. The exponent may reference other existing elenents by indirect addressing. This result could in turn be simplified by the RTERM routine.

It may of course be necessary to raise a simple factor group to an explicit power. This could be done by decomposing a simple group of m variable into m+1 composite factors, each of which consists of a simple item raised to an explicit power. It is, however, also possible to raise the complete simple item to the explicit power by creating a single composite factor consisting of the simple group all of which is affected by the explicit exponent. This latter course is

followed in AIDS.

REDUCE operates by maintaining a stack for the exponent value as a simple algebraic expression. Each factor in the term or composite factor may contribute its own exponent value which must of course by removed before passing to the next factor. For example assume that the term 3 * A ** 2 * B * (A - B) ** 2 * (A + D) ** 3 were to be raised to the power X - Y. An exponent stack consisting essentially of a simple algebraic expression with the single term (X - Y), which is itself a simple algebraic expression, is created. The first sub element of the term which is a simple factor with explicit exponents is raised to an exponent power by creating a composite factor with a single simple data element. The composite factor is set to type explicit. The exponent value is moved from the exponent stack to the resultant area. The next factor (A - B) ** 2 is a simple algebraic expression with explicit exponent. A control word pointing to the simple exponent of value "2 " and designated as type term is added to the exponent stack. The control word describing the exponent stack is also updated to reflect the increased length and element count. The local exponent value is removed from the exponent stack, by changing the exponent controll word, after the factor has been moved off to the result area. This process is repeated until all

sub element factors have been dealt with.

This routine is particularly useful in cases where structure is created and manipulated by routines other than the AIDS routines for performing operation on the data structure elements. However, AIDS does use this facility when expanding a polynomial to an integer power.

A limited amount of structural simplification is provided by the MOVELEM routine. All indirect addressing links are removed from the subelements of the data element being moved. The result of course is stored in canonical format.

Substitution

Substitution involves the replacement of one symbolic entity by another. The entity being replaced is usually a variable although it could be any algebraic element.

This process is essentially scan driven and requires searching for the element to be replaced. At present AIDS provides the facility of replacing a symbolic variable by any other algebraic data element. As a symbolic variable is always stored in a simple element a complete simple element must be removed and in most cases replaced by a composite element. This is done in AIDS by creating indirect address control words in an existing

structure which point to the replacement element. A simplifying pass is then used to move the structure to a resultant area.

Substitution is performed in AIDS by the routine SUBST which has as argument the addresses of the variable for replacement, the referenced simple algebraic expression and the resultant area. Even though replacement of complete elements is not provided it can readily be designed by using existing facilities in AIDS.

Chapter VII

Differentiation and Integration

Differentiation and integration constitute two of the more complex algebraic functions of algebraic manipulation. Both of course are required when dealing with the formal solution of differential equations. The procedures for performing these operations are recursive in nature with procedural complexity related to the sophistication of the associated data structure. Each procedure can be defined in terms of primative arithmetic and pattern matching operations as well as predefined transformations.

Differentiation, as is to be expected, is decidedly the simpler of the two processes. Initial attempts (Refs. 4, 5) at symbolic differentiation were crude and based on simple data structures. With the availability of LISP, which is well suited to this type of operation, more sophisticated and powerful schemes emerged (Ref. 3). Differentiation can also be readily achieved with string processing languages such as SNOBOL.

Symbolic integration, however, does present more of a challenge. The first symbolic integration program with any claim to generality was SAINT (Symbolic Automatic Integrator - Ref. 6). This was written in LISP but did

not have sufficient flexibility to permit the solution of ordinary differential equations in a practical algebraic manipulation system. This was followed by SIN (Symbolic Integration - Ref. 8), also written in LISP, which used the rational function package of MATHLAB (Ref. 7). SIN in turn was used to write SOLDIER (Ref. 8) for the solution of first order, first degree ordinary differential equations. Both systems make extensive use of pattern matching.

Even though SAINT and SIN are both relatively powerful they can be very time consuming because of their heuristic approach. Integration is perhaps best suited to a highly interactive environment where user direction and involvement can be used to reduce the complexity of the operation.

Differentiation and Integration in AIDS

Differentiation

A comprehensive differentiation facility is incorporated within AIDS with many of the associated routines making use of the recursive facilities. A design objective in AIDS has been to provide data structures which permit the realisation of complex operations in a natural and efficient manner. This can be demonstrated by the relative ease with which the normally difficult operation of differentiation can be achieved.

Differentiation of any element (including simple algebraic expressions) is performed as a single pass in-line operation to produce resultant elements. As these elements may not be simplified a further simplification pass is required which also moves the elements to a resultant area. No other re-organisation of the data elements is necessary even though the operation is inherently highly recursive.

There are 3 levels of routine in AIDS involving differentiation of:

1. simple algebraic expressions
2. terms - simple or composite
3. functions

All of the above are essentially recursive except for the

case of simple term elements.

Differentiation of a simple algebraic expression is performed by the DIFFSAE routine which accepts as parameters the address of the simple algebraic expression control word, the address where the result is to be stored and the address of the variable of differentiation. Each term is referenced in turn and a test is made to determine whether it is of type simple or composite before control is transferred to the appropriate routine. The control word of the resultant simple algebraic expression is automatically updated after return of control from the lower level differentiation routines. This routine is essentially recursive since a term within a simple algebraic expression may itself be totally or in part another simple algebraic expression. A final simplification pass is required to remove redundant terms from the unsimplified structure created in the work area.

The recursive routine DIFFC differentiates a composite term with respect to a given variable. The parameters involved are the addresses of the composite term, result area and variable of differentiation as well as the address of the simple algebraic expression control word (of which the resultant term is a part) to be updated. Differentiation is performed according to the formula:

$$\frac{d}{dx}(f_1 f_2 f_3 \dots f_n) = f_2 f_3 \dots\dots f_n \frac{df_1}{dx} + f_1 f_3 f_4 \dots f_n \frac{df_2}{dx} + \dots\dots + f_1 f_2 f_3 \dots f_{n-1} \frac{df_n}{dx}$$

Each term in the result requires creating a null term control word followed by the differentiation of the factor associated with the term. If the result of the differentiation is zero this term is ignored and the routine proceeds to create the next term. If a non zero result is produced indirect address factor control words are stored with the term to reference the remaining factors. Both the term control word and the simple algebraic expression control word are appropriately updated. The resultant term is simplified and moved to its proper result area.

In performing differentiation of any element in the above process a test is made for type. If the element is a simple factor, control is passed to the routine for performing differentiation of simple elements. If a simple algebraic expression is encountered a further test for explicit exponent is made else control is immediately passed to the DIFFSAE routine. Should an exponent exist it is referenced as an indirectly addressed factor of the term before control is passed to DIFFSAE. When control is returned the control word for the expression, if the result is non zero, is modified to specify an explicit exponent which is then formed by performing a subtraction of a term with value 1/1 from the existing exponent.

For the case where the element to be differentiated is a function designator control is passed to DIFFFD. This

routine is required to identify the function and perform the necessary operation. For trigometric functions this can be performed through tables. Further if a function is affected by an explicit exponent it is dealt with in the same manner as the analogous situation for simple algebraic expressions. DIFFFD is recursive.

The most common requirement is to differentiate simple elements which can be of either implicit or explicit exponent format. This is essentially a very fast operation as it involves a scan to determine the presence of the variable of differentiation. When the referenced element has implicit exponents and a match is found the element is copied less the variable into the resultant area. For explicit exponents the appropriate exponent value is decreased by 1/1 before moving it. A resultant exponent value of zero requires removing the variable during copying. To give an indication of the result a completion code is returned in R15, zero indicating a valid result and non zero that the result itself is zero.

Considerably more power can be added to this system by catering to functional dependence of one variable on another. For this, further scans would be required to search for dependent variables. This operation would then involve replacing the simple element with a composite element involving factors which are themselves derivatives.

Integration

No attempt has as yet been made to implement general integration schemes involving all the algebraic data elements of AIDS. However, a routine (INTEG) does exist for integrating a simple element with respect to a specified variable of integration. This function is similar in operation to differentiating a simple element except that in this case it may be necessary to expand a simple element from implicit to explicit exponent format. Again no functional dependence is permitted at this stage.

Integration of complicated algebraic expressions can be achieved by first removing parentheses and then integrating on a term by term basis. It is hoped to be able to attempt more ambitious schemes in the future.

Chapter VIII

CLAM Algebraic Interpreter

CLAM, an acronym for Command Language for Algebraic Manipulation is a simple interpretive scheme designed to illustrate the facilities of AIDS. It consists of a number of basic commands available to the user for performing a limited class of algebraic operations. The system can readily be extended to include other commands by interfacing to the existing facilities of AIDS.

The interpreter uses some of the recogniser routines from the syntax analyser which recognises simple algebraic expressions. Each command statement is free format hence blanks may appear anywhere. An implementation restriction requires that the total length of a command must not exceed 400 characters, excluding blanks, and the command must be terminated by a semi colon.

CLAM was written by interfacing to existing AIDS facilities. The power of AIDS is not limited to writing simple algebraic interpreters as it is expected to be able to produce sophisticated algebraic compilers from AIDS.

Description of CLAM

The following further syntactic entities are used in the definition of CLAM:

1. <name> : : = <variable identifier>
2. <name list> : : = <name> | <name list> " , " <name>

A name may reference a symbolic variable or a simple algebraic expression. The unique identification of a user is also by a name.

A command statement consists of a command identifier followed by an operand list and terminated by a semicolon. The following is a list of commands available in CLAM:

1. USER < name > " ; "

This is the first command of a CLAM program and identifies a user to the system. CLAM then allocates him a private data space for tables, work areas, stack areas, etc.

2. FINI < name > " ; "

FINI signals the end of processing for a given user. For terminal operation a user count is kept which when it goes to zero causes control to return to the operating system. A user's data space is relinquished by FINI.

3. LET < name > "=" < simple algebraic expression > " ; "

LET permits a simple algebraic expression and its associated name to be known to CLAM. The simple algebraic expression is recognised, converted to internal format, and catalogued.

4. PRINT < name list > " ; "

PRINT causes a print out of all simple algebraic expressions referenced in the name list. Names which reference variables are printed with their current value.

5. SET < name > "=" < number > "," < name >
   "=" < number > . . . ","

Symbolic variables are assigned numeric values by means of the SET command. Any number of variables may be assigned values through a single SET command.

6. SAVE <name list > ";"

The expressions referenced by names in the name list are relegated to backing store. They may be restored by either the command RESTORE or a reference to an simple algebraic expression during processing.

8. RESTORE < name list > ";"

All the referenced simple algebraic expressions in the name list are moved from backing store to main store.

9. DISPLAY { NAME TABLE
   STRUCTURE TABLE } ";"

The contents of either the NAME or STRUCTURE table is printed out.

10. EVALUATE <name list> ";"

The simple algebraic expressions referenced in the name list are evaluated for numeric results.

11. SUBSTITUTE < name > "=" < simple algebraic expression >
   "IN " < name > ";"

The first name value references a variable in the simple algebraic expression specified by the second name value. The referenced variable is replaced by the symbolic value of the simple algebraic expression.

12. SIMPLIFY <name list> " ; "

Each simple algebraic expression referenced by a variable in the name list is simplified and stored in main core in canonical format.

13. DIFFERENTIATE <name> " WRT " <simple variable> " ; "

The simple algebraic expression referenced is differentiated with respect to a simple variable

14. EXPAND <name list> " ; "

This command is used to reduce a simple algebraic expression to a polynomial of simple terms.

15. INTEGRATE <name> " WRT " <simple variable> " ; "

Integration of the named simple algebraic expression is performed with respect to the simple variable specified on a term basis. The simple algebraic expression must first be expanded.

An example of a CLAM program is given in Appendix VII.

Chapter IX

Summary

AIDS has been designed from the premise that algebraic operations can be defined on algebraic data structure elements in much the same way that arithmetic operations are defined on arithmetic data. Further it is possible to identify processes and facilities which are essential for manipulating symbolic data. Some of these constitute primitive operations for symbolic processing and could well be incorporated within the hardware.

The usefulness of AIDS lies in the ability to define simple algebraic data elements, composed of primitive (arithmetic and symbolic) components, which can be combined dynamically with algebraic structural information to form composite algebraic data elements. The data elements are organised to model algebraic expressions. By manipulating the data elements through algorithmic schemes, algebraic operations are realised. Desirable hardware features can be associated with specific algebraic operations.

The accessing mechanism involves indirect addressing to any level which is a standard hardware feature on many machines. Indirect addressing is usually associated with

a specific bit position in the instruction. Obviously for an AIDS implementation it would be desirable to define the control word format in such a way that the indirect address bit coincides with the hardware indirect address bit.

Data management in AIDS makes extensive use of stacks. Again this facility is found on many machines and is becoming fashionable with new machine architecture. This stack facility must be capable of dealing with variable length segments. Stack maintenance should be done by the hardware except for stack overlays. A stack area full or empty condition signalled by a hardware interrupt can be used to initiate a stack area save or stack area restore operation. It would also be desirable to provide a store protection mechanism through hardware rather than through software as in AIDS. For existing data elements the length field can be used to define the upper bound with the control word as base.

In AIDS it is desirable to be able to manipulate data elements in single operations. When multiplying two simple elements of the same exponent format the operation is well defined and involves a concatenation, a rational arithmetic multiplication and the creation of new control word. Reducing a simple element with explicit exponents to a minimal form is also a well defined operation. When a simple element has implicit format,

scans through the symbolic components for multiple instances of the same component could set a condition code to signify that the element requires expansion to explicit exponents which must then be followed by a reduce operation. The expansion from implicit to explicit format is itself well-defined. All of the above operations could be implemented as interruptable micro-programmed sequences. It is possible that rational arithmetic, including the greatest common denominator algorithm, could be realised through micro code, however because of its potentially explosive nature when the resultant denominator is small condition code setting would certainly be required.

Future enhancement

It has always been the desire of the author to develop the concept for ultimate use in man-machine interactive systems. It is obvious that many processes such as factoring can be best realised through user interaction. At the same time it may be informative for the user to have a graphical representation of a function on which to base further decisions. Identification and changes from the display can be readily implemented through association lists using indirect addressing. A display with keyboard terminal is an obvious candidate for the AIDS facility.

An obvious enhancement is the representation of complex numbers. This can be achieved by essentially

adding another dimension (and using another bit in control word) to each simple element. Consideration should also be given to extending the concept to ordered n-triples for tensor applications.

It is further hoped that the AIDS facility can be used to devise languages for teaching and demonstrating concepts in some areas of mathematics.

References

1. Iliffe, J.K. Basic Machine Principles, Macdonald & Co. (London) 1968

2. Iliffe, J.K. Elements of BLM, Computer Journal August (1969)

3. Woolridge, D. An Algebraic Simplification Program in LISP, Stanford Artificial Intelligence Project, Memorandum 11, December 1963

4. Kahrimanian, H.G. Analytical Differentiation by Digital Computer, M.A. Thesis, Temple University Philadelphia, Pa. May 1953

5. Symbolic Work on High Speed Computers, Project Report No. 4, Dartmouth Mathematical Project, Dartmouth College, N.H. June 1954

6. Slagle, J.R. A Heuristic Program that Solves Symbolic Integration Problems, Journal ACM 10, 4 p. 507 - 520

7. Engleman, C MATHLAB, Proceedings AFIPS 65, FJCC November 1965 p. 413 - 422

8. Moses, J. Symbolic Integration, Project MAC report MAC-TR-47, December 1967

9. Brown, W.S. The ALPAK System, Bell Systems Technical Journal Vol. XLII (1963) p. 2681

10. Tobey, R.G., R.J. Bubrow, S.N. Zilles, Automatic Simplification in FORMAC, Proceedings Fall Joint Computer Conference 1963

11. Brown,W.S., J.P. Hyde, B.A. Tague, ALPAK System, Bell Systems Technical Journal XLIII No. 2 (1964), p. 785 - 804

12. Collins, G.E. "PM A System for Polynomial Manipulation", Communications ACM 9 (August 1966) p. 578 - 589

13. Brown, W.S. A Language for Symbolic Algebra on a Digital Computer, Proceedings IBM Scientific Comp. Symposium on Computer Aided Experimentation, October 1965

14. Bond, E.R. FORMAC Share Program General Library R21BM0016

15. Sammet, J.E. and E.R. Bond Introduction to FORMAC Trans. IEEE on Elect. Comp. August 1964

16. Sammet, J.E. Survey of Formula Manipulation, Communications ACM Vol. 9, No. 8 (August 1966)

17. Sammet, J.E. An Annoted Description Based Bibliography on the Use of Computers for Non-numerical Mathematics, Computing Reviews Vol. 7, No. 4 (July 1966) p. B1 - B31

18. Engeli, M. Design and Implementation of an Algebraic Processor, Report Institute für angewandte Mathematik der E TH, Zurich April 1966

19 Engeli, M. SYMBAL - User's Manual, Report: The University of Texas at Austin, June 1968

20. Sibley, E.H. The Engineering Assistant: Design of Symbol Manipulation System, Technical Report CONCOMP, The University of Michigan August 1967

21. Fenichel, R.R. An On-line System for Algebraic Manipulation, Report MAC-TR-35, Prog. MAC, MY-T December 1966

22. Standish, T.A. A Data Definition Facility for Programming Languages, Ph.D. Thesis, Carnegie Institute of Technology, Pittsburgh, Pa. May 1967

23. Perlis, A.J., R. Iturriaga An Extension of ALGOL for Manipulating Formulae, Communications ACM 7, February 1964

24. Griswold, R.E., J.F. Poage, I.P. Polansky The SNOBOL 4 Programming Language, Prentice Hall, 1968

25. McCarthy, J., et al., LISP 1.5 Programmer's Manual M.I.T. Press, Cambridge, Mass. 1962

26. Newell, A., H.S. Kelly IPL-V Manual, Prentice Hall 1964

27. Wirth, N., H. Weber EULER, A Generalisation of ALGOL, and Its Definition, Part I Comm. ACM Vol. 9 No. 1 January 1966, Part II Comm. ACM Vol. 9 No. 2 February 1966

28. Ross, D. et al, AED-O Programming Manual, AED-O User's Kit, Electronic Systems Laboratory, MIT, Cambridge, Mass.

29. Wirth, N., C.A.R. Hoare A Contribution to the Development of ALGOL, Communications ACM Vol. 9, No. 6 June 1966

30. Iliffe, J.K., J.G. Jodeit A Dynamic Storage Allocation Scheme, Computer Journal 5, 200 (1962)

31. Roos, D. ICES System Design, M.I.T. Press Cambridge, Mass. 1967

32. Iliffe, J.K., G.F. Coulouris Notes on the "Machine Interface" Presented at NATO Advanced Study Institute on Architecture and Design of Digital Computers, August 1969

33. System 360 - 2928 Array Processor, IMB Manual Ref. A24-3519

34. APL/360 Primer IBM Manual

Appendix I

Syntax For Algebraic Expressions

1. Basic Symbols

<basic symbol> ::= <letter> | <digit> | <delimiter>

1.1 <letter> ::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

1.2 <digit> ::=0|1|2|3|4|5|6|7|8|9

1.3 <delimiter> ::= <operator> | <separator> | <bracket>

1.3.1 <operator> ::= <arithmetic operator> | <relational operator> | <logical operator>

1.3.1.1 <arithmetic operator> ::= <add operator> | <multiplication operator> | <exponentiation operator>

1.3.1.1.1 <add operator> ::=+ | -

1.3.1.1.2 <multiplication operator> ::=* | /

1.3.1.1.3 <exponentiation operator> ::=**

1.3.1.2 <relational operator> ::= <|<=|=|>=|>|¬=

1.3.1.3 <logical operator> ::= ≡|&|ı|¬

1.3.2 <separator> ::=,|.|:|;

1.3.3 <bracket> ::= (|)

2. Identifiers

2.1 <letter digit string> ::= <letter> | <letter digit string> <letter> | <letter digit string> <digit>

2.2 <identifier> ::= <letter digit string>

3. Numbers

<number> ::= <integer> | <rational> | <real>

3.1 <unsigned integer> ::= <digit> | <unsigned integer> <digit>

3.2; <integer> ::= <unsigned integer> | <addop> <unsigned integer>

3.3 <rational number> ::= <integer> " / " <integer>

3.4 <real> ::= <integer> " . " | " . " <unsigned integer> | <integer> " . " <unsigned integer>

4. Variables

<variable> ::= <simple variable> | <subscripted variable>

4.1 <variable identifier> ::= <identifier>

4.2 <simple variable> ::= <variable identifier>

4.3 <array identifier> ::= <identifier>

4.4 <subscripted variable> ::= <array identifier> " (" <subscript list> ") "

4.5 <subscript list> ::= <subscript expression> | <subscript list> " , " <subscript expression>

4.6 <subscript expression> ::= <simple algebraic expression>

5. Function Designator

5.1 <function designator> ::= <variable identifier> " (" <parameter list> ") "

5.2 <parameter list> ::= <parameter> | <parameter list> " , " <parameter>

5.3 <parameter> ::= <identifier> | <simple algebraic expression>

6. Algebraic Expressions

6.1 <variable exponent pair> ::= <simple variable> " ** " <number> | <simple variable> " ** " " (" <number> ") "

6.2 <simple variable group> ::= <simple variable> | <simple variable group> " * " <simple variable>

6.3 <variable group> ::= <variable exponent pair> | <variable group> <multiplication operator> <variable exponent pair>

6.4 <simple factor group> ::= <simple variable group> | <number> " * " <simple variable group> | <number>

6.5 <factor group> ::= <variable group> | <number> <multop> <variable group>

6.6 <primary> ::= <simple factor group> | <factor group> | <function designator> | " (" <simple algebraic expression> ") "

6.7 <factor> ::= <primary> | <factor> " ** " <primary>

6.8 <term> ::= <factor> | <term> <multiplication operator> <factor>

6.9 <simple algebraic expression> ::= <term> | <addop> <term> | <simple algebraic expression> <addop> <term>

Appendix II

Summary of AIDS macros

1. Recursive Facility Macros

   1. RSAVE R1= 0, R2 =14, STACK=0

      saves the consecutive sequence of registers as specified by keyword parameters R1 and R2 on the stack specified by the STACK keyword parameter. (Default values will cause registers 0 to 14 to be saved on stack 0).

   2. RRETURN R1= 0, R2 =14, STACK=0

      restores the consecutive sequence of registers as specified by keyword parameters R1 and R2 on the stack specified by the STACK parameter. The default values are identical to those in RSAVE.

   3. STACKSEG ADDRESS, LENGTH, STACK=0

      saves the consecutive sequence of words of length (in words) specified by the positional parameter LENGTH from store location specified by the positional parameter ADDRESS on the general purpose stack specified by the keyword parameter STACK as an extension of the existing stack segment. The default value for the STACK parameter is stack 0.

4. POPSEG ADDRESS, LENGTH, STACK=0

restores a single sequence of words of length (in words) specified by the positional parameter LENGTH from store location specified by the positional parameter ADDRESS from the general purpose stack specified by the keyword parameter STACK. The same default value as for the STACKSEG macro is assumed.

5. STACK ADDRESS, LENGTH, STACK=1

saves the sequence of words of length (in words) as specified by the LENGTH positional parameter from store location specified by the positional parameter ADDRESS as a single data stack segment on the data stack specified by the keyword parameter STACK. The default value for the STACK parameter is stack 1.

6. POP ADDRESS, LENGTH, STACK=1

restores the sequence of words of length (in words) as specified by the LENGTH positional parameter from the last stack segment on the stack specified by the STACK keyword parameter to the store location specified by the positional parameter ADDRESS as a single stack segment.

II. Data Element Referencing Macros

1. NEXTD RP=2, RD=3

   loads the register specified by the keyword parameter RD with the address of the first sub-element of the element whose address is contained in the register specified by the keyword parameter RP. The default values for RP and RD are 2 and 3 respectively.

2. NEXTE RP=2, RE=3

   loads the register specified by the keyword parameter RE with the address of the next equivalent element to the element whose address is contained in the register specified by the keyword parameter RP. The default values for RP and RD are 2 and 3 respectively.

3. ELEMENT PR=2, ER=3

   loads the register specified by the keyword parameter ER with the final address of the element whose address is contained in the register specified by the keyword parameter PR. The default values for PR and ER are 2 and 3 respectively. There can be any number of levels of indirectness.

4. EXPONENT RP=2, RE=3

loads the register specified by the keyword parameter RE with the address of the exponent element for the element whose address is contained in the register specified by the keyword parameter PR. The default values for RP and RE are 2 and 3 respectively.

5. FUNCTION RP=2, RE=3

loads the register specified by the keyword parameter RF with the address of the name element for the function whose address is contained in the register specified by RP.

6. ALL1 FUNC, RP=2, RE=3, R1=4

executes the function whose symbolic address is specified by the positional parameter FUNC for all sub-elements of the element whose address is contained in the register specified by the keyword parameter RP. FUNC must return control through register 14. The keyword parameter RE specifies the register to be used for holding the address of each sub-element, before transfering control to the function procedure. R1 is the keyword parameter which specifies the count register for the sub-elements. The default values for RP, RE and R1 are 2,3 and 4 respectively.

7. ALL2 FUNC, RP=2, RE1=3, RE2=4, R1=5, R2=6

executes the function whose symbolic address is specified by the positional parameter FUNC for all sub-elements of the sub-elements whose address is contained in the register specified by the keyword parameter RP. (This macro, for example, can be used to access all of the factors of all of the terms in a simple algebraic expression). Keyword parameters RE1 and RE2 specify the registers to be used to store the addresses of the first and second level sub-elements respectively. R1 and R2 are keyword parameters for specifying the respective count registers.

III. Utility Macros

1. MOVE LR=5, FROMR=6, TOR=7

moves the consecutive sequence of bytes from the main store address contained in the register specified by the keyword parameter FROMR to the main store address, contained in the register specified by keyword parameter TOR. The number of bytes to be moved is found in the register specified by the keyword parameter LR. The registers specified by FROMR and TOR are updated after the operation, while the value in the register specified by LR goes to zero.

2. DEFINE

defines the hexadecimal equivalence of a single byte for a number of common symbols used in AIDS. The following values are defined on the first byte of a control word:

a) SIMPLE defines the simple/composite bit

b) IMPLICIT defines the implicit/explicit exponent bit

c) INDIRECT defines the indirect addressing bit

d) PRESENCE defines the presence bit

e) TERM defines the one's complement of the 2 bit values for "term" (note "term" has a bit field value of 00)

f) FACTOR defines the 2 bit value for "factor"

g) SAE defines the 2 bit value for "simple algebraic expression"

h) FD defines the 2 bit value for "function designator"

The following possible type fields are defined:

i) STI simple term implicit

j) STE simple term explicit

k) CTI composite term implicit

l) CTE composite term explicit

m) SFI simple factor implicit

n) SFE simple factor explicit

o) CFI composite factor implicit

p) SEI simple algebraic expression implicit

q) SEE simple algebraic expression explicit

r) FDI function designator implicit

s) FDE function designator explicit

IV. Syntax Analyzer Macros

1. STRTABLE NS=00, D=60, LT=00, AO=00, MO=00,
   RO=00, LO=00, S=00

   create a 256 byte vector and assigns the value (hexadecimal) of the appropriate type, as defined by the keyword parameters, to each byte whose offset is the decimal equivalent of the hexadecimal EBCDIC value. The following types are defined:

type	description	graphic symbol
a) NS	non valid symbol	non graphic symbol
b) D	digit	0 - 9
c) LT	upper case letters	A - Z
d) AO	add operator	+ , -
e) MO	multiplication operator	* , /
f) RO	relational operator	< , >, =
g) LO	logical operator	¬, &, |
h) S	separators	, ; : .
i) DL	delimitors	(,), ' "
j) SS	special symbols	_, ?, &, %, @, #
k) BLK	blank	

2. BOT TYPE, ADDRESS

causes abranch to the symbollic location specified by the positional parameter ADDRESS only in the next character in the source string being examined is of type as specified by the positional parameter TYPE. The typecan be any one of the types listed in the STRTABLE macro above.

3. BNT TYPE, ADDRESS

causes a branch to the symbolic location specified by the positional parameter ADDRESS only if the next character in the source string being examined is not of the type as specified by the positional parameter TYPE.

Appendix III

Control Word Formats

1. Format 1 - standard format

bit position	value	definition
0	0	simple element
	1	composite element
1	0	implicit exponent
	1	explicit exponent
2 - 3	00	term
	01	factor
	10	simple algebraic expression
	11	function designators
4	0	in-line addressing
	1	indirect addressing (use format 2)
5	0	locally defined element
	1	externally defined element
6 - 15		number of sub elements
16 - 31		overall length of element or offset to next equivalent element

2. Format 2 - indirect addressing format

bits	definition
0 - 3	as in format 1
4 1	indirect addressing
8 - 31	new address of control word

Appendix 1V

Description of Condition Byte after Logical Tests

1. TESTESI - test for equivalence of 2 simple elements.[1]

bit position	definition
0	constants differ in value
1	exponent types differ
2	exponents are inverse values
3	variables do not match

2. TESTECI - test for equivalence of 2 composite elements

bit position	definition
0	constants of simple elements differ in value
1	sub elements do not match
2	exponents differ in value
3	number of sub elements differ
4	no simple sub element

1 a value of 1 in bit position for condition to hold

Appendix V

Table Formats

1. Structure Name Table

All entries of 8 bytes each

a) header work for table

number of entries in table

b) table entries

bytes			
1 - 4	name of structure		
5	status byte		
	bit 0	0	in main core
		1	on secondary storagedevice
	bit 1	0	inactive
		1	active

format 2

6	identification number of saved record
7 - 8	length of saved record in words

format 1

5 - 8	address in core of structure

2. Name value table

Each entry is 8 bytes long as follows:

1 - 4	name of variable
5 - 8	value in floating point representation

Appendix VI

Summary of CLAM Commands

The following further syntactic types are used in the definition of CLAM:

<name> :: <variable identifier>

<name list> :: <name> <name list> " , " <name>

Command Structure

Command Operands " ; "

Blanks are permitted anywhere

CLAM Commands

1. SAVE <name list> " ; "
2. RESTORE <name list> " ; "
3. DELETE <name list> " ; "
4. EVALUATE <name list> " ; "
5. SET { <name> " " <number> " , " <name> " = " <number> ..... } " ; "
6. DIFFERENTIATE <name> "WRT" <simple variable> " ; "
7. INTEGRATE <name> "WRT" <simple variable> " ; "
8. EXPAND <name list> " ; "
9. DISPLAY { NAME TABLE / STRUCTURE TABLE } " ; "
10. SUBSTITUTE <name> " = " <simple algebraic expression> " IN " <name> " ; "
11. PRINT <name list> " ; "
12. SIMPLIFY <name list> " ; "
13. USER <name> " ; "
14. LET <name> " = " simple algebraic expression " ; "
15. FINI <name> " ; "

```
IEF298I  SYNAN     SYSOUT=B.

//SYNAN JOB 'LINDERS',MSGLEVEL=1
//    EXEC PROC=ASMFCLG,PARM.LKED='XREF,LET,LIST'
XXASM      EXEC  PGM=IEUASM,PARM=LOAD,REGION=50K                          00020000
XXSYSLIB  DD     DSNAME=SYS1.MACLIB,DISP=SHR                              00040000
XXSYSUT1  DD     UNIT=SYSSQ,SPACE=(1700,(400,50))                         00060000
XXSYSUT2  DD     UNIT=SYSSQ,SPACE=(1700,(400,50))                         00080000
XXSYSUT3  DD     UNIT=(SYSSQ,SEP=(SYSUT2,SYSUT1,SYSLIB)),                X00100000
XX               SPACE=(1700,(400,50))                                    00120000
XXSYSPRINT DD    SYSOUT=A                                                 00140000
XXSYSPUNCH DD    SYSOUT=B                                                 00160000
XXSYSGO   DD     DSNAME=&LOADSET,UNIT=SYSSQ,SPACE=(80,(100,50)),         X00180000
XX               DISP=(MOD,PASS)                                          00200000
//ASM.SYSIN DD *
IEF236I ALLOC. FOR SYNAN     ASM
IEF237I SYSLIB    ON 131
IEF237I SYSUT1    ON 132
IEF237I SYSUT2    ON 136
IEF237I SYSUT3    ON 130
IEF237I SYSPRINT  ON 131
IEF237I SYSPUNCH  ON 132
IEF237I SYSGO     ON 135
IEF237I SYSIN     ON 130
```

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

SYMBOL	TYPE	ID	ADDR	LENGTH	LD ID
SYNAN	SD	01	000000	001E62	
INAREA	ER	02			
CMD	ER	03			
ADDSTR	ER	04			
STRTAB	ER	05			
WA	ER	06			
MOVELEM	ER	07			
DSAVE	ER	08			
DELETES	ER	09			
EVAL	ER	0A			
CRLSTR	ER	0B			
GETCOR	ER	0C			
DIFFSAE	ER	0D			
PRTSAE	ER	0E			
RETCOR	ER	0F			
INTSER	ER	10			
DISPLAY	ER	11			
SUBST	ER	12			
NEWU	ER	13			
HLC	ER	14			
INSERTN	ER	15			
EXPANDS	ER	16			
INSERTS	ER	17			
SERIES	ER	18			
TERM	ER	19			
PURGE	ER	1A			
USERAREA	ER	1B			
LDS	LD		000F08		01
IDENT	LD		000F56		01
VI	LD		000F6C		01
SV	LD		000F82		01
VEP	LD		000FBE		01
SVG	LD		001030		01
VG	LD		001090		01
FD	LD		00111A		01
PLIST	LD		0012A6		01
PARM	LD		0012E6		01
SEG	LD		0012FC		01
FG	LD		0013B6		01
NUMBER	LD		00148E		01
JI	LD		0014C8		01
RATIONAL	LD		00157C		01
REAL	LD		0015C8		01
REALL	LD		0016B8		01
PRIMARY	LD		00173E		01
FACTOR	LD		0017FA		01
TERM	LD		001A62		01
SAE	LD		00134C		01
STAK	ER	1C			
POPD	ER	1D			
MESSAGES	ER	1E			
CNVRL	ER	1F			
INITIAL	LD		001EBC		20
MESSAGES	SD	20	001E68	000118	
USERA	ER	21			

UNIVERSITY OF WATERLOO

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                        F01FEB69   4/01/70

                                    1 *
                                    2          MACRO
                                    3 &NAME    STACKSEG &ADDRESS,&LENGTH,&STACK=0
                                    4 &NAME    L        15,=V(STAKSEG)
                                    5 *    NOTE THAT LENGTH PARAMETER IS ADDRESS OF LENGTH VALUE
                                    6          LA       14,M&SYSNDX+8
                                    7          LA       1,M&SYSNDX
                                    8          BR       15
                                    9          CNOP     0,4
                                   10 M&SYSNDX DC       AL1(&STACK)
                                   11          DC       AL3(&ADDRESS)
                                   12          DC       A(&LENGTH)
                                   13          MEND
                                   14          MACRO
                                   15 &NAME    POPSEG   &ADDRESS,&LENGTH,&STACK=0
                                   16 &NAME    L        15,=V(POPSG)
                                   17 *    NOTE THAT LENGTH PARAMETER IS ADDRESS OF LENGTH VALUE
                                   18          LA       14,N&SYSNDX+8
                                   19          LA       1,N&SYSNDX
                                   20          BR       15
                                   21          CNOP     0,4
                                   22 N&SYSNDX DC       AL1(&STACK)
                                   23          DC       AL3(&ADDRESS)
                                   24          DC       A(&LENGTH)
                                   25          MEND
                                   26          MACRO
                                   27 &NAME    STACK    &ADDRESS,&LENGTH,&STACK=1
                                   28 &NAME    L        15,=V(STAK)
                                   29 *    NOTE THAT LENGTH PARAMETER IS ADDRESS OF LENGTH VALUE
                                   30          LA       14,&ADDRESS
                                   31          ST       14,RSA&STACK
                                   32          LA       14,&LENGTH
                                   33          ST       14,RSA&STACK+4
                                   34          LA       1,RSA&STACK
                                   35          MVI      0(1),X'0&STACK'
                                   36          BALR     14,15
                                   37          MEND
                                   38          MACRO
                                   39 &NAME    POP      &ADDRESS,&LENGTH,&STACK=1
                                   40 &NAME    L        15,=V(POPD)
                                   41 *    NOTE THAT LENGTH PARAMETER IS ADDRESS OF LENGTH VALUE
                                   42          LA       14,&ADDRESS
                                   43          ST       14,RSA&STACK
                                   44          ST       14,RSA&STACK
                                   45          LA       14,&LENGTH
                                   46          ST       14,RSA&STACK+4
                                   47          LA       1,RSA&STACK
                                   48          MVI      0(1),X'0&STACK'
                                   49          BALR     14,15
                                   50          MEND
                                   51          MACRO
                                   52 &NAME    RSAVE    &R1=0,&R2=14,&NR=X'0F',&STACK=0
                                   53 &NAME    BALR     15,0            ESTABLISH BASE
                                   54          USING    *,15
                                   55          L        15,=V(NASSA&STACK)
```

UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

```
LOC  OBJECT CODE     ADDR1 ADDR2  STMT    SOURCE STATEMENT                                              F01FEB69   4/01/70

                                    56           L       15,0(0,15)
                                    57           STM     &R1,&R2,4(15) SAVE REGISTERS
                                    58           MVI     3(15),&NR
                                    59           BALR    15,0
                                    60           USING   *,15
                                    61           MVI     A&SYSNDX,X'0&STACK'
                                    62           LA      14,A&SYSNDX+4
                                    63           L       15,A&SYSNDX
                                    64           BR      15
                                    65  A&SYSNDX DC      V(RSTACK)
                                    66           USING   *,15
                                    67           MEND
                                    68           MACRO
                                    69  &NAME    RRETURN &R1=0,&R2=14,&STACK=0
                                    70  &NAME    BALR    15,0
                                    71           USING   *,15
                                    72           L       15,=V(CSSA&STACK)
                                    73           L       15,0(0,15)
                                    74           LM      &R1,&R2,4(15) RESTORE REGISTERS
                                    75           BALR    15,0
                                    76           USING   *,15
                                    77           MVI     B&SYSNDX,X'0&STACK'
                                    78           L       15,B&SYSNDX
                                    79           BR      15
                                    80  B&SYSNDX DC      V(USTACK)
                                    81           MEND
                                    82           MACRO
                                    83  &NAME    MOVE    &LR=5,&FROMR=6,&TOR=7
                                    84  *     SPECIFIED IN REGISTER FROMR TO LOCATION SPECIFIED IN REGISTER TOR
                                    85  &NAME    CH      &LR,=H'256'
                                    86           BL      Q&SYSNDX          LESS THAN 256 BYTES TO MOVE
                                    87           MVC     0(256,&TOR),0(&FROMR)  MOVE BLOCK OF 256 BYTES
                                    88           LA      &FROMR,256(0,&FROMR)   INCREMENT REGISTER BY 256
                                    89           LA      &TOR,256(0,&TOR)
                                    90           SH      &LR,=H'256'
                                    91           B       *-26
                                    92  Q&SYSNDX LTR     &LR,&LR
                                    93           BZ      P&SYSNDX
                                    94           BCT     &LR,*+4
                                    95           EX      &LR,*+12
                                    96           LA      &LR,1(0,&LR)  RE-INCREMENT COUNT
                                    97           B       P&SYSNDX
                                    98           MVC     0(1,&TOR),0(&FROMR)  MOVE PARTIAL BLOCK
                                    99  P&SYSNDX LA      &TOR,0(&LR,&TOR)
                                   100           LA      &FROMR,0(&LR,&FROMR)
                                   101           MEND
                                   102           MACRO
                                   103  &NAME    NEXTD   &RP=2,&RD=3
                                   104  &NAME    ELEMENT PR=&RP,ER=&RD
                                   105           L       &RD,4(0,&RD)  PICK UP NEXT INLINE CW
                                   106           MEND
                                   107           MACRO
                                   108  &NAME    NEXTE   &RP=2,&RE=3
                                   109  &NAME    TM      0(&RP),INDIRECT
                                   110           BZ      B&SYSNDX
```

UNIVERSITY OF WATERLOO

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                             F01FEB69   4/01/70

                                  111          TM      0(&RP),X'C0'    TEST FOR EXPLICIT COMPOSITE
                                  112          BO      C&SYSNDX
                                  113          LA      &RE,4(0,&RP)
                                  114          B       D&SYSNDX+2
                                  115 C&SYSNDX LA      &RE,8(0,&RP)
                                  116          B       D&SYSNDX+2
                                  117 B&SYSNDX LH      &RE,2(0,&RP)
                                  118          SLA     &RE,2(0)
                                  119 D&SYSNDX AR      &RE,&RP
                                  120          MEND
                                  121          MACRO
                                  122 &NAME    ELEMENT &PR=2,&ER=3
                                  123 &NAME    LR      &ER,&PR
                                  124          TM      0(&ER),INDIRECT
                                  125          BZ      *+12
                                  126          L       &ER,0(0,&ER)
                                  127          B       *-12
                                  128          MEND
                                  129          MACRO
                                  130 &NAME    EXPONENT &RP=2,&RE=3
                                  131 &NAME    TM      0(&RP),INDIRECT
                                  132          BZ      V&SYSNDX
                                  133 *    IF INDIRECT EXPONENT MUST BE NEXT ELEMENT
                                  134          LA      &RE,4(0,&RP)
                                  135          ELEMENT PR=&RE,ER=&RE
                                  136          B       Z&SYSNDX+2
                                  137 V&SYSNDX LR      14,&RP          SAVE POINTER
                                  138          LH      15,0(0,&RP)
                                  139          N       15,MASK1
                                  140          LA      15,1(0,15)
                                  141          NEXTD   RP=&RP,RD=&RE
                                  142          B       W&SYSNDX
                                  143 X&SYSNDX NEXTE   RP=&RP,RE=&RE
                                  144 W&SYSNDX LR      &RP,&RE
                                  145          ELEMENT PR=&RE,ER=&RE
                                  146          BCT     15,X&SYSNDX
                                  147 Z&SYSNDX LR      &RP,14
                                  148          MEND
                                  149          MACRO
                                  150 &NAME    FUNCTION &RP=2,&RF=3
                                  151 &NAME    NEXTD   RP=&RP,RD=&RF
                                  152          MEND
                                  153          MACRO
                                  154 &NAME    ALL1    &FUNC,&RP=2,&RE=3,&R1=4
                                  155 &NAME    ELEMENT PR=&RP,ER=&RP
                                  156 *    EXECUTE FUNCTION FOR ALL FIRST LEVEL ITEMS
                                  157          L       &R1,0(0,&RP)
                                  158          N       &R1,MASK1       NO. OF ELEMENTS AT THIS LEVEL
                                  159          NEXTD   RP=&RP,RD=&RE
                                  160          B       L&SYSNDX
                                  161 K&SYSNDX NEXTE   RP=&RP,RE=&RE
                                  162 L&SYSNDX LR      &RP,&RE
                                  163          ELEMENT PR=&RE,ER=&RE
                                  164          L       15,=A(&FUNC)
                                  165          BALR    14,15
```

UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT    SOURCE STATEMENT                                   F01FEB69   4/01/70
                                  166            BCT     &R1,K&SYSNDX
                                  167            MEND
                                  168            MACRO
                                  169  &NAME     ALL2    &FUNC,&RP=2,&R1=5,&R2=6,&RE1=3,&RE2=4
                                  170  &NAME     ELEMENT RP=&RP,ER=&RP
                                  171  *    MACRO EXECUTES FUNCTION FOR ALL ELEMENTS AT SECOND LEVEL DOWN
                                  172            L       &R1,0(0,&RP)
                                  173            N       &R1,MASK1       NO. OF FIRST LEVEL ELEMENTS
                                  174            NEXTD   RP=&RP,RD=&RE1
                                  175            B       N&SYSNDX
                                  176  M&SYSNDX  NEXTE   RP=&RP,RE=&RE1
                                  177  N&SYSNDX  LR      &RP,&RE1
                                  178  *
                                  179            ELEMENT PR=&RE1,ER=&RE1
                                  180            TM      0(&RE1),SIMPLE
                                  181            BZ      Q&SYSNDX
                                  182            L       &R2,0(0,&RE1)
                                  183            N       &R2,MASK1
                                  184            TM 0(&RE1),IMPLICIT
                                  185            BZ      *+8
                                  186            LA      &R1,1(0,&R1)    INCREMENT COUNT BY 1
                                  187            TM      0(&RE1),X'30'   TEST FOR FD
                                  188            BNO     *+8
                                  189            LA      &R1,1(0,&R1)    INCREMENT COUNT BY 1
                                  190  *
                                  191            NEXTD   PR=&RE1,ER=&RE2
                                  192            B       P&SYSNDX
                                  193  O&SYSNDX  NEXTE   RP=&RE1,RE=&RE2
                                  194  P&SYSNDX  LR      &RE1,&RE2
                                  195            ELEMENT PR=&RE2,ER=&RE2
                                  196  Q&SYSNDX  L       15,=A(&FUNC)
                                  197            BALR    14,15
                                  198            BCT     &R2,O&SYSNDX
                                  199            BCT     &R1,M&SYSNDX
                                  200            MEND
                                  201            MACRO
                                  202            DEFINE
                                  203  SIMPLE    EQU     X'80'            ELEMENTARY OR COMPOSITE ITEM
                                  204  IMPLICIT  EQU     X'40'            IMPLICIT OR EXPLICIT FORMAT FOR EXPONENTS
                                  205  TERM      EQU     X'30'            NOTE -MUST BRANCH ON ZERO FOR TERM
                                  206  FACTOR    EQU     X'10'            TYPE FACTOR
                                  207  SAE       EQU     X'20'            TYPE SIMPLE ALGEBRAIC EXPRESSION
                                  208  FD        EQU     X'30'            TYPE FUNCTION DESIGNATOR
                                  209  INDIRECT  EQU     X'08'            INDIRECT ADDRESSING
                                  210  PRESENCE  EQU     X'04'            EXTERNALLY DEFINED ELEMENT
                                  211  *          TYPES
                                  212  STI       EQU     X'00'            SIMPLE TERM IMPLICIT
                                  213  STE       EQU     X'40'            SIMPLE TERM EXPLICIT
                                  214  CTI       EQU     X'80'            COMPOSITE TERM IMPLICIT
                                  215  CTE       EQU     X'C0'            COMPOSITE TERM EXPLICIT
                                  216  SFI       EQU     X'10'            SIMPLE FACTOR IMPLICIT
                                  217  SFE       EQU     X'50'            SIMPLE FACTOR EXPLICIT
                                  218  CFI       EQU     X'90'            COMPOSITE FACTOR IMPLICIT
                                  219  CFE       EQU     X'D0'            COMPOSITE FACTOR EXPLICIT
                                  220  SEI       EQU     X'20'            SIMPLE ALGEBRAIC EXPRESSION IMPLICIT
```

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT    SOURCE STATEMENT                                          F01FEB69   4/01/70

                                  221 SEE      EQU      X'60'            SIMPLE ALGEBRAIC EXPRESSION EXPLICIT
                                  222 FDI      EQU      X'B0'            FUNCTION IMPLICIT
                                  223 FDE      EQU      X'F0'            FUNCTION EXPLICIT
                                  224 FNI      EQU      X'B0'            FUNCTION IMPLICIT
                                  225 FNE      EQU      X'F0'            FUNCTION EXPLICIT
                                  226 *   ENVIRONMENT BITS
                                  227 ESAE     EQU      X'80'
                                  228 EFD      EQU      X'40'
                                  229 MD       EQU      X'10'
                                  230 SFGE     EQU      X'08'
                                  231 FGE      EQU      X'04'
                                  232 SAEE       EQU    X'01'
                                  233          MEND
                                  234          MACRO
                                  235 &NAME    TEST     &N=0,&FROM=*,&TO=*+4000
                                  236 &NAME    ST       1,X&SYSNDX
                                  237          SNAP     DCB=DEBUG,ID=&N,PDATA=(REGS),STORAGE=(&FROM,&TO)
                                  238          L        1,X&SYSNDX
                                  239          B        X&SYSNDX+4
                                  240 X&SYSNDX DS       1F
                                  241          MEND
                                  242 *
                                  243 *   MACRO DEFINITIONS FOR SYNTAX ANALYSER
                                  244          MACRO
                                  245 &NAME    SETENV   &TYPE
                                  246 &NAME    OI       NVSTATUS,&TYPE
                                  247          MEND
                                  248          MACRO
                                  249 &NAME    CLEARENV &TYPE
                                  250 &NAME    XI       NVSTATUS,&TYPE
                                  251          MEND
                                  252          MACRO
                                  253 &NAME    CLER     &ADDRESS,&LENGTH,&C=C' '
                                  254 &NAME    MVI      &ADDRESS,&C    PUT CHARACTER IN FIRST POSITION
                                  255          MVC      &ADDRESS(1)+LENGTH-1,&ADDRESS
                                  256          MEND
                                  257          MACRO
                                  258 &NAME    BIFTYPE  &TYPE,&ADDRESS,&AR=2
                                  259 &NAME    CLI      0(&AR),&TYPE   TEST FOR TYPE
                                  260          BE       &ADDRESS
                                  261          MEND
                                  262          MACRO
                                  263 &NAME    NEXTCHAR &R=3
                                  264 &NAME    LA       &R,1(0,&R)      INCREMENT STRING POINTER BY 1
                                  265          MEND
                                  266          MACRO
                                  267 &NAME    BNT      &TYPE,&ADDRESS
                                  268 &NAME    TRT      0(1,3),CODESTR OBTAIN FUNCTION BYTE FOR CHARACTER
                                  269          EX       2,*+12          REFERENCED BY R3- INSERTED IN R2
                                  270          BNE      &ADDRESS
                                  271          B        *+8
                                  272          CLI      &TYPE,X'00'
                                  273          MEND
                                  274          MACRO
                                  275 &NAME    BOT      &TYPE,&ADDRESS
```

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                              F01FEB69   4/01/70

                                 276 &NAME   TRT      0(1,3),CODESTR OBTAIN FUNCTION BYTE FOR CHARACTER
                                 277         EX       2,*+12         REFERENCED BY R3- INSERTED IN R2
                                 278         BE       &ADDRESS
                                 279         B        *+8
                                 280         CLI      &TYPE,X'00'
                                 281         MEND
                                 282         MACRO
                                 283 &NAME   STRTABLE &NS=00,&D=00,&LT=00,&AO=00,&MO=00,&RO=00,&LO=00,&S=00, X
                                                      &DL=00,&SS=00,&BLK=00
                                 284 *    NS-NON VALID SYMBOLS HAVE VALUE      &NS       NON GRAPHIC SYMBOL
                                 285 *    D-DIGIT                              &D        0-9
                                 286 *    LT-UPPER CASE LETTERS                &LT       A-Z
                                 287 *    AO-OPERATORS                         &AO       +,-
                                 288 *    MO-MULTIPLY OPERATORS                &MO       *,/
                                 289 *    RO-RELATIONAL OPERATORS              &RO       =, ,
                                 290 *    LO-LOGICAL OPERATORS                 &LO       &, ,
                                 291 *    S-SEPARATORS                         &S        . F   ,
                                 292 *    DL-DELIMITORS                        &DL       (,),',
                                 293 *    SS-SPECIAL SYMBOLS                   &SS        , , ,$, , , ,
                                 294 *   BLK-BLANK
                                 295 *
                                 296 * ALL PARAMETERS SPECIFY HEXADECIMAL VALUES
                                 297 *
                                 298 *          TABLE VALUE             SYMBOL      DECIMAL      TYPE
                                 299 &NAME      DC    64X'&NS'                      0-63         NS
                                 300            DC    XL1'&BLK'         64          &BLK
                                 301            DC    9X'&NS'           65-73       NS
                                 302            DC    XL1'&SS'                      74           SS
                                 303            DC    XL1'&S'           .           75           S
                                 304            DC    XL1'&RO'          <           76           RO
                                 305            DC    XL1'&DL'          (           77           DL
                                 306            DC    XL1'&AO'          +           78           AO
                                 307            DC    XL1'&LO'                      79           LO
                                 308            DC    XL1'&LO'          &           80           LO
                                 309            DC    9X'&NS'                       81-89        NS
                                 310            DC    XL1'&SS'                      90           SS
                                 311            DC    XL1'&SS'          $           91           SS
                                 312            DC    XL1'&MO'          *           92           MO
                                 313            DC    XL1'&DL'          )           93           DL
                                 314            DC    XL1'&S'                       94           S
                                 315            DC    XL1'&LO'                      95           LO
                                 316            DC    XL1'&AO'          -           96           AO
                                 317            DC    XL1'&MO'          /           97           MO
                                 318            DC    9X'&NS'                       98-106       NS
                                 319            DC    XL1'&S'           ,           107          S
                                 320            DC    XL1'&SS'                      108          SS
                                 321            DC    XL1'&SS'                      109          SS
                                 322            DC    XL1'&RO'                      110          RO
                                 323            DC    XL1'&SS'                      111          SS
                                 324            DC    10X'&NS'                      112-121      NS
                                 325            DC    XL1'&S'                       122          S
                                 326            DC    XL1'&SS'                      123          SS
                                 327            DC    XL1'&SS'                      124          SS
                                 328            DC    XL1'&DL'          '           125          DL
                                 329            DC    XL1'&RO'          =           126          RO
```

UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                F01FEB69   4/01/70
                                 330            DC     XL1'&DL'                   127       DL
                                 331            DC     XL1'&NS'                   128       NS
                                 332            DC     64X'&NS'                   129-182   LOWER CASE
                                 333            DC     9X'&LT'          A-I       193-201   LT
                                 334            DC     7X'&NS'                    202-208   NS
                                 335            DC     9X'&LT'          J-R       209-217   LT
                                 336            DC     8X'&NS'                    218-225   NS
                                 337            DC     8X'&LT'          S-Z       226-233   LT
                                 338            DC     6X'&NS'                    234-239   NS
                                 339            DC     10X'&D'          0-9       240-249   D
                                 340            DC     6X'&NS'                    250-255   NS
                                 341 *
                                 342 *   FUNCTION VALUES
                                 343 *
                                 344 DIGIT  DC     XL1'&D'
                                 345 LETTER DC     XL1'&LT'
                                 346 ADDOP  DC     XL1'&AO'
                                 347 MULTOP DC     XL1'&MO'
                                 348 RELOP  DC     XL1'&RO'
                                 349 LOGOP  DC     XL1'&LO'
                                 350 SEP    DC     XL1'&S'
                                 351 DELIM  DC     XL1'&DL'
                                 352 SPEC   DC     XL1'&SS'
                                 353 NONSYM DC     XL1'&NS'
                                 354 BLANK  DC     XL1'&BLK'
                                 355 RBRAC  DC     1C'('
                                 356 LBRAC  DC     1C')'
                                 357 ASTERISK DC   1C'*'
                                 358          MEND
                                 359          MACRO
                                 360 &NAME    BD       &ADDRESS
                                 361 &NAME    EX       2,*+8
                                 362          B        *+8
                                 363          CLI      DIGIT,X'00'
                                 364          BE       &ADDRESS
                                 365          MEND
                                 366 *
                                 367          MACRO
                                 368 &NAME    BLT      &ADDRESS
                                 369 &NAME    STC      2,*+5            BRANCH IF LETTER
                                 370          EX       2,*+8
                                 371          B        *+8
                                 372          CLI      LETTER,X'00'
                                 373          MEND
                                 374 *
                                 375          MACRO
                                 376 &NAME    BBLK     &ADDRESS
                                 377 &NAME    EX       2,*+8
                                 378          B        *+8
                                 379          CLI      BLANK,X'00'
                                 380          BE       &ADDRESS
                                 381          MEND
                                 382 *
                                 383          MACRO
                                 384 &NAME    BNAO     &ADDRESS
```

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

```
LOC    OBJECT CODE      ADDR1 ADDR2  STMT   SOURCE STATEMENT                                          F01FEB69   4/01/70

                                     385 &NAME   EX      2,*+8
                                     386         B       *+8
                                     387         CLI     ADDOP,X'00'
                                     388         BNE     &ADDRESS
                                     389         MEND
                                     390 *
                                     391         MACRO
                                     392 &NAME   BNMO    &ADDRESS         BRANCH IF NOT MULT OP
                                     393 &NAME   STC     2,*+5
                                     394         CLI     MULTIO,X'00'
                                     395         BNE     &ADDRESS
                                     396         MEND
                                     397 *
                                     398 *   COMMAND LIST FUNCTION
                                     399         MACRO
                                     400         CLF     &L,&FUNC
                                     401         LA      14,&L.(0,0)
                                     402         LR      3,4
                                     403 CM&SYSNDX LA    3,0(14,3)        R3 POINTS TO STRING
                                     404         LA      2,CI&SYSNDX
                                     405         L       15,=A(STAKAP)
                                     406         BALR    14,15
                                     407         L       15,=A(SV)
                                     408         LA      10,SYNTAX+6
                                     409         ST      11,AG1
                                     410         LA      11,4095(0,10)
                                     411         LA      11,1(0,11)
                                     412         BALR    14,15
                                     413 CI&SYSNDX L   11,AG1
                                     414         LTR     0,0
                                     415         BZ      CE1
                                     416         LR      14,0
                                     417         LA      3,0(14,3)        UPDATE R3 TO GET BY VI
                                     418 *   PERFORM OPERATION
                                     419         BAL     14,&FUNC
                                     420         LA      14,1(0,0)
                                     421         CLI     0(3),C','        IS NEXT CHARACTER A COMMA
                                     422         BNE     N
                                     423         B       CM&SYSNDX
                                     424         MEND
                                     425 *   END OF MACRO DEFINITIONS
000000                               426 SYNAN   CSECT
000000 90EC D00C          0000C      427 CMDINT  STM     14,12,12(13)
000004 05B0                          428         BALR    11,0
000006                               429         USING   *,11
                                     430 *
000006 4120 BE2A          00E30      431         LA      2,CISA
00000A 5020 D008          00008      432         ST      2,8(0,13)
00000E 50D0 2004          00004      433         ST      13,4(0,2)
000012 18D2                          434         LR      13,2
000014 58F0 BD3A          00D40      435         L       15,=A(INITIAL)
000018 05EF                          436         BALR    14,15
                                     437 *   OPEN INPUT DATA SET
                                     438         OPEN    (IN,(INPUT))
00001A 0700                          439+          CNOP  0,4
```

UNIVERSITY OF WATERLOO

```
IEF285I   SYS1.MACLIB                                   KEPT
IEF285I   VOL SER NOS= MFT222.
IEF285I   SYS70091.T113346.RF000.SYNAN.R0000001         DELETED
IEF285I   VOL SER NOS= MATH44.
IEF285I   SYS70091.T113346.RF000.SYNAN.R0000002         DELETED
IEF285I   VOL SER NOS= MATH33.
IEF285I   SYS70091.T113346.RF000.SYNAN.R0000003         DELETED
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.SF000.SYNAN.R0000004         SYSOUT
IEF285I   VOL SER NOS= MFT222.
IEF285I   SYS70091.T113346.SF000.SYNAN.R0000005         SYSOUT
IEF285I   VOL SER NOS= MATH44.
IEF285I   SYS70091.T113346.RF000.SYNAN.LOADSET          PASSED
IEF285I   VOL SER NOS= MATH22.
IEF285I   SYS70091.T113346.RF000.SYNAN.S0000006         SYSIN
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.RF000.SYNAN.S0000006         DELETED
IEF285I   VOL SER NOS= MFT111.
XXLKED    EXEC  PGM=IEWL,PARM=(XREF,LET,LIST,NCAL),REGION=96K,            C00220000
XX              COND=(8,LT,ASM)                                            00240000
XXSYSLIN  DD    DSNAME=&LOADSET,DISP=(OLD,DELETE)                          00260000
XX        DD    DDNAME=SYSIN                                               00280000
XXSYSLMOD DD    DSNAME=&GOSET(GO),UNIT=SYSDA,SPACE=(1024,(50,20,1)),      X00300000
XX              DISP=(MOD,PASS)                                            00320000
XXSYSUT1  DD    UNIT=(SYSDA,SEP=(SYSLIN,SYSLMOD)),SPACE=(1024,(50,20))     00340000
XXSYSPRINT DD  SYSOUT=A,DCB=(,BLKSIZE=121)                                 00360000
//LKED.SYSLIB DD DSNAME=SYS1.FORTLIB,DISP=SHR
//LKED.SYSIN DD *
IEF236I ALLOC. FOR SYNAN    LKED
IEF237I SYSLIN   ON 135
IEF237I          ON 130
IEF237I SYSLMOD  ON 130
IEF237I SYSUT1   ON 131
IEF237I SYSPRINT ON 136
IEF237I SYSLIB   ON 130
```

UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

F44-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LET,LIST
VARIABLE OPTIONS USED - SIZE=(49152,10240) DEFAULT OPTION(S) USED
****GO DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET

CROSS REFERENCE TABLE

CONTROL SECTION NAME	ORIGIN	LENGTH	ENTRY NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
SYNAN	00	1E62								
			LDS	F08	IDENT	F56	VI	F6C	SV	F82
			VEP	FBE	SVG	1030	VG	1090	FD	111A
			PLIST	12A6	PARM	12E6	SFG	12FC	FG	13B6
			NUMBER	148E	UI	14C8	RATIONAL	157C	REAL	15C8
			REALL	16B8	PRIMARY	173E	FACTOR	17FA	TERM	1A62
			SAE	1B4C						
MESSAGES	1E68	118								
			INITIAL	1EBC						
TABLES	1F80	210								
			INSERTN	1F80	INSERTS	1FEC	DELETEN	205E	DELETES	20A2
			NAMVAL	20EA	ADDSTR	2138				
INTSER	2190	12C								
			CNPRRL	227C						
DISPLAY	22C0	100								
STORE	23C0	9F54								
			GETCOR	23C0	RETCOR	24EA				
ALGOTS	C318	2254								
			TESTESI	C318	TESTECI	C43C	HLC	C790	MOVELEM	CB2C
			REDUCE	CDFE	MOVECWS	CF48	ASAES	D256	SSAES	D3FA
			TMULT	D5CC	TDIV	D624	DSAES	D670	MSAES	D7AE
			COMPRESS	DA40	EXPAND	DB2A	PRDIV	DBC6	PRMULT	DBF0
			PRSUB	DC3C	PRADD	DC52				
PRINT	E570	8E7								
			PRTSAE	E570						
SERIES	EE58	12								
			PURGE	EE58						
NEWU	EE70	404								
CRLSTR	F278	D0								
DIFF	F348	71A								
			DIFFSAE	F348	DIFFS	F7DA	MATCHVAR	F922	INTEG	F97C
RSTACK	FA68	754								
			USTACK	FB66	STAKSEG	FBF4	STAK	FC7C	POPD	FCF2
			POPSG	FD6E	DSAVE	FDCA	DGET	FEA0	INAREA	10040
EVAL	101C0	AF2								
			EXPANDS	10588						
SUBST	10CB8	20E								
CNVRL	10EC8	198								
ATX	11060	15A								
RTI	111C0	1F0								
STAK0	113B0	7D0								
STAK1	11B80	7D0								
STAK2	12350	7D0								

UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
USERAREA	12B20	1248	CSSA0	12B24	STACK0	12B24	NASSA0	12B28	CSSA1	12B44
			NASSA1	12B48	CSSA2	12B64	NASSA2	12B68		
USERA	13D68	E38	NAMTAB	13F14	STRTAB	14558				
IHCFIXPI*	14BA0	14F	FIXPI	14BA0						
IHCFRXPR*	14CF0	183	FRXPR	14CF0						
IHCSLOG *	14E78	1BA	ALOG10	14E78	ALOG	14E94				
IHCSEXP *	15038	1B0	EXP	15038						
IHCECOMH*	151E8	F31	IBCOM	151E8	FDIOCS	152A4	INTSWTCH	16106		
IHCCOMH2*	16120	545	SEQDASD	16380						
IHCERRM *	16668	5AC	ERRMON	16668	IHCERRE	16680				
IHCFCVTH*	16C18	1175	ADCON	16C18	FCVAOUTP	16CC2	FCVLOUTP	16D52	FCVZOUTP	16EA2
			FCVIOUTP	1722E	FCVEOUTP	17730	FCVCOUTP	1794A	INT6SWCH	17C33
IHCEFNTH*	17D90	512	ARITH	17D90	ADJSWTCH	180FC				
IHCEFIOS*	182A8	111C	FIOCS	182A8	FIOCSBEP	182AE				
IHCUOPT *	193C8	300								
IHCETRCH*	196C8	28E	IHCTRCH	196C8	ERRTRA	196D0				
IHCUATBL*	19958	88								
WA	199E0	3E80								
RCWV	1D860	7D0								
XWAS	1E030	FA0	CMD	1E030	CWVSAE	1E350	VFCWS	1E670	VTCWS	1E800

UNIVERSITY OF WATERLOO

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
D44	INAREA	RSTACK	D4C	CMD	XWAS
D60	ADDSTR	TABLES	D64	STRTAB	USERA
D68	WA	WA	D6C	MOVELEM	ALGUTS
D70	DSAVE	RSTACK	D74	DELETES	TABLES
D78	EVAL	EVAL	D7C	CRLSTR	CRLSTR
D84	GETCOR	STORE	D88	DIFFSAE	DIFF
D8C	PRTSAE	PRINT	D90	RETCOR	STORE
D94	INTSER	INTSER	D98	DISPLAY	DISPLAY
DA0	SUBST	SUBST	DA8	NEWU	NEWU
DAC	HLC	ALGUTS	DB4	INSERTN	TABLES
DB8	EXPANDS	EVAL	DBC	INSERTS	TABLES
DC4	SERIES	SERIES	DD0	PURGE	SERIES
DD4	USERAREA	USERAREA	D40	MESSAGES	MESSAGES
D50	MESSAGES	MESSAGES	1CA8	WA	WA

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
1CA0	HLC	ALGUTS
1CB8	POPD	RSTACK
1CC0	CNVRL	CNVRL
2188	USERA	USERA
22B8	CNVRL	CNVRL
23AC	STRTAB	USERA
23B4	NAMTAB	USERA
C460	RSTACK	RSTACK
CF6C	RSTACK	RSTACK
CDAC	USTACK	RSTACK
C780	NASSA0	USERAREA
C78C	CSSA0	USERAREA
D24C	RCWV	RCWV
D614	RETCOR	STORE
DE08	RETCOR	STORE
E594	RSTACK	RSTACK
EADC	RSTACK	RSTACK
E990	USTACK	RSTACK
EE38	NASSA0	USERAREA
F270	MESSAGES	MESSAGES
F344	RTI	RTI
F4A4	RSTACK	RSTACK
F5F4	USTACK	RSTACK
FA38	CSSA0	USERAREA
FA40	STAK	RSTACK
FA48	PRSUB	ALGUTS
FA54	PRADD	ALGUTS
FA5C	EXPAND	ALGUTS
10030	MESSAGES	MESSAGES
10398	RSTACK	RSTACK
10470	USTACK	RSTACK
1059C	ATX	ATX
105B0	CNPRRL	INTSER
106A0	VTCWS	XWAS
10C88	WA	WA
10C98	GETCOR	STORE
10CA0	RETCOR	STORE
10CDC	RSTACK	RSTACK
10EB0	NASSA0	USERAREA
10EC0	GETCOR	STORE
11250	FIXPI	IHCFIXPI
12B28	STAK0	STAK0
12B34	STAK0	STAK0
12B48	STAK1	STAK1
12B54	STAK1	STAK1
12B68	STAK2	STAK2
12B74	STAK2	STAK2
14C80	IHCERRM	IHCERRM
14E04	IHCERRM	IHCERRM
14DFC	EXP	IHCSEXP
14FE0	IHCERRM	IHCERRM
15160	IHCERRM	IHCERRM

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
1CB4	STAK	RSTACK
1CBC	MESSAGES	MESSAGES
1EB8	USERA	USERA
2278	INTEG	DIFF
23A8	USERA	USERA
23B0	MESSAGES	MESSAGES
23BC	CRLSTR	CRLSTR
CB50	RSTACK	RSTACK
C770	USTACK	RSTACK
D23C	USTACK	RSTACK
D240	NASSA0	USERAREA
D248	CSSA0	USERAREA
D608	GETCOR	STORE
DDFC	GETCOR	STORE
DE1C	MESSAGES	MESSAGES
E76C	RSTACK	RSTACK
E684	USTACK	RSTACK
EC0C	USTACK	RSTACK
EE3C	CSSA0	USERAREA
F340	USERA	USERA
F36C	RSTACK	RSTACK
F47C	USTACK	RSTACK
FA30	NASSA0	USERAREA
FA3C	VTCWS	XWAS
FA44	POPD	RSTACK
FA4C	MOVELEM	ALGUTS
FA58	PRDIV	ALGUTS
10018	STACK0	USERAREA
101E4	RSTACK	RSTACK
1033C	USTACK	RSTACK
10598	NASSA0	USERAREA
105A0	CSSA0	USERAREA
105B4	NAMVAL	TABLES
106A4	MOVELEM	ALGUTS
10C94	CWVSAE	XWAS
10C9C	REDUCE	ALGUTS
10CA8	MESSAGES	MESSAGES
10D84	USTACK	RSTACK
10EB8	CSSA0	USERAREA
110F0	FRXPR	IHCFRXPR
12B24	STAK0	STAK0
12B30	STAK0	STAK0
12B44	STAK1	STAK1
12B50	STAK1	STAK1
12B64	STAK2	STAK2
12B70	STAK2	STAK2
14C84	IBCOM	IHCECOMH
14E00	IBCOM	IHCECOMH
14DF8	ALOG	IHCSLOG
14FA8	IBCOM	IHCECOMH
15164	IBCOM	IHCECOMH
152A4	SEQDASD	IHCCOMH2

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
1600C	ADCON	IHCFCVTH	16004	FIOCS	IHCEFIOS
16010	ARITH	IHCEFNTH	16030	ADJSWTCH	IHCEFNTH
15FB8	IHCUOPT	IHCUOPT	1602C	IHCUOPT	IHCUOPT
16014	FCVEOUTP	IHCFCVTH	16018	FCVLOUTP	IHCFCVTH
1601C	FCVIOUTP	IHCFCVTH	16020	FCVCOUTP	IHCFCVTH
16024	FCVAOUTP	IHCFCVTH	16028	FCVZOUTP	IHCFCVTH
15FB4	IHCERRE	IHCERRM	15FE4	IHCCOMH2	IHCCOMH2
15FE8	IHCERRM	IHCERRM	15FBC	IHCCOMH2	IHCCOMH2
15FC0	IHCCOMH2	IHCCOMH2	15FC4	IHCCOMH2	IHCCOMH2
15FC8	IHCCOMH2	IHCCOMH2	16289	IHCECOMH	IHCECOMH
1628C	IHCECOMH	IHCECOMH	164C5	IHCECOMH	IHCECOMH
164D5	IHCECOMH	IHCECOMH	164E5	IHCECOMH	IHCECOMH
16C04	IHCUOPT	IHCUOPT	16C08	IBCOM	IHCECOMH
16C0C	IHCTRCH	IHCETRCH	16C10	FIOCSBEP	IHCEFIOS
17BF4	IBCOM	IHCECOMH	17BF0	IHCERRM	IHCERRM
1814C	IBCOM	IHCECOMH	18150	INTSWTCH	IHCECOMH
180F8	INT6SWCH	IHCFCVTH	180F4	IHCUOPT	IHCUOPT
18158	ADCON	IHCFCVTH	18154	FIOCS	IHCEFIOS
181C4	IHCERRM	IHCERRM	18400	IHCERRM	IHCERRM
1848C	IHCUATBL	IHCUATBL	18498	IBCOM	IHCECOMH
1983C	IBCOM	IHCECOMH	19840	ADCON	IHCFCVTH
19844	FIOCSBEP	IHCEFIOS			

ENTRY ADDRESS 00
TOTAL LENGTH 1EFD0

UNIVERSITY OF WATERLOO

```
IEF285I   SYS70091.T113346.RF000.SYNAN.LOADSET          DELETED
IEF285I   VOL SER NOS= MATH22.
IEF285I   SYS70091.T113346.RF000.SYNAN.S0000009         SYSIN
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.RF000.SYNAN.S0000009         DELETED
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.RF000.SYNAN.GOSET            PASSED
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.RF000.SYNAN.R0000007         DELETED
IEF285I   VOL SER NOS= MFT222.
IEF285I   SYS70091.T113346.SF000.SYNAN.R0000008         SYSOUT
IEF285I   VOL SER NOS= MATH33.
IEF285I   SYS1.FORTLIB                                  KEPT
IEF285I   VOL SER NOS= MFT111.
XXGO      EXEC  PGM=*.LKED.SYSLMOD,COND=((8,LT,ASM),(4,LT,LKED))         00380000
//GO.SYSUDUMP DD SYSOUT=A
//GO.MESSAGES DD SYSOUT=A
//GO.SAVEAREA DD UNIT=2314,SPACE=(CYL,20),VOLUME=SER=MATH44
//GO.ALGEBRAS DD *
IEF236I ALLOC. FOR SYNAN    GO
IEF237I PGM=*.DD ON 130
IEF237I SYSUDUMP ON 131
IEF237I MESSAGES ON 131
IEF237I SAVEAREA ON 132
IEF237I ALGEBRAS ON 130

        USER L1
        LET E1=A**2+2.0*B
        LET E2=(A+2.0*B)**2+4.1*A**3-9.1+E1
        PRINT E1,E2
 E1=(1/1*A**(2/1)+2/1*B)
 E2=((1/1*A+2/1*B)**(2/1)+41/10*A**(3/1)+91/10+E1)
        SET A=2.0, B=3.0
        EVALUATE E2
 VALUE OF    E2       =97.7
        DELETE B
        SUBSTITUTE B=A+3   IN E2
        PRINT E1,E2
 E1=(1/1*A**(2/1)+2/1*B)
 E2=((3/1*A+6)**(2/1)+41/10*A**(3/1)-91/10+E1)
        EXPAND E2
 (9/1*A**(2/1)+36*A+451/10+41/10*A**(3/1)+E1)
        DIFFERENTIATE E2  WRT A
 (18/1*A+36/1+123/10*A**(2/1))
        SAVE E2
        DISPLAY STRUCTURE TABLE
 STRUCTURE E1  IN CORE
 STRUCTURE E2   ON DISC
        DISPLAY NAME TABLE
 VALUE OF     A       =2.0
       FINI
```

UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

```
IEF285I   SYS70091.T113346.RF000.SYNAN.GOSET             PASSED
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.SF000.SYNAN.R0000010          DELETED
IEF285I   VOL SER NOS= MFT222.
IEF285I   SYS70091.T113346.SF000.SYNAN.R0000011          SYSOUT
IEF285I   VOL SER NOS= MFT222.
IEF285I   SYS70091.T113346.RF000.SYNAN.R0000012          DELETED
IEF285I   VOL SER NOS= MATH44.
IEF285I   SYS70091.T113346.RF000.SYNAN.S0000013          SYSIN
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.RF000.SYNAN.S0000013          DELETED
IEF285I   VOL SER NOS= MFT111.
IEF285I   SYS70091.T113346.RF000.SYNAN.GOSET             DELETED
IEF285I   VOL SER NOS= MFT111.
```

UNIVERSITY OF WATERLOO

UNIVERSITY OF WATERLOO