ON A NEW TOOL IN ARTIFICIAL INTELLIGENCE RESEARCH.

AN ASSOCIATE MEMORY, PARALLEL PROCESSING

LANGUAGE, AMPPL-II.

by

Nicholas V. Findler and Wiley R. McKinzie*
Department of Computer Science
State University of New York at Buffalo

ABSTRACT

One of the remarkable things about human information processing is the way we store and retrieve information. The human associative memory is a device that has been only rather crudely approximated with computers in flexibility, self-modifiability, and complexity of associations.

In this paper, we describe a computer language, AMPPL-II, which simulates a machine with an extensive, variable size associative memory and parallel processing capability. It was to be made fairly machine-independent and is, therefore, embedded in a high-leve1 algebraic language. The version outlined here is built as an extension of SLIP (Symmetric List Processing Language), itself beinq embedded in FORTRAN IV.

The expected range of applicability of AMPPL-II covers a rather wide area. Candidates in the non-numerical field are game playing, picture processing and image identification, encoding-decoding, simulation of cognitive behavior, multikey information retrieval, language analysis, sorting, merging, collating, query systems, and so on. Numerical applications would be matrix calculations, solution of partial differential equations, computation of auto- and cross-correlation functions, signal correction, etc.

Stimulated by this languaae, new computational techniques and algorithms may, hopefully, be developed. These, on one hand, and technological advances, on the other, may render the hardware implementation of an AMPPL-like machine economical.

*N.V.F. designed AMPPL and is responsible for the mistakes contained in this paper. W. R. McK. implemented the language on the CDC 6400. Teiji Furugori's contribution to the latter is also acknowledged. Further, Peter Mullor and Charles Bergenstock worked on an IBM 7044 implementation of AMPPL-I.

KJ.Y WORDS AND PiiPASI s

Associative menory, parallel processing, symbol manipulation, list processina, extension of algebraic lanuuages, information structures.

CR CATLOORIES

3.60, 3.69, 4.10, 4.22

INTRODUCTION

There is an obvious interrelation between the level of complexity of the problems that are ootentially solvable on computers, and the power of available software and hardware. Further dcvclonment in proaramming systems and languages enables us to attack problems that have so far been beyond the sphere of practicability. List processing and string manipulating languages, for examnle, have opener[1] un new vistas in many research fields, probably most significantly in Artifical Intelligence.

We describe in this paper an approach to memory and information processing, radically different from those of the traditional von Neumann-type computers. An Associative Memory, Parallel Processing Language, AMPPL-II*, is introduced, which simulates a computer built in hardware. (We point out that some of the facilities available in AMPPL-II would not be present in the engineer's implementation. However, they do not "cost" much in programmina effort and in computing time, and they may also be handy to have. We have, therefore, decided to include a few "non-generic" features.)

* The first version, AMPPL-I, was reported on in [1,2] and was implemented but not completely debugqed for the IBM 7044. The presently described language, running on the CDC 6400, has improved dynamic memory allocation facilities and more powerful instructions concerning Relations (cf. Section II/C).

It should be noted here that there are many references in the literature to hardware and software efforts that are aimed at objectives similar to ours here. The engineering activity can be divided into two, rather distinct categories, rirst, very expensive "black boxer," have been constructed and attached to snecial or general purpose computers to perform some of the functions v/e shall discuss later. Second, small associative memory units have been included in computer systems for specific purposes, such as the paging operation with large-scale time-shared machines.

\ number of programming systems have also been developed, most of them directed towards specific, some of them towards more general applications. V.'e only wish to single out here the work of Feldman and Rovner [3,4,5], and of Dodd, Beach and Rossol [6]. Their objectives are very similar to ours, although the respective techniques are different. Experience will show which of these languages is more powerful, easier to program and debug in.

Excellent surveys, covering the field in question up to the middle of 1966, can be found in [7,8]. For the sake of completeness, we mention a few more, some left out of the above surveys, some of more recent vintage [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20].

I. SOME BASIC CONCEPTS

In contrast with conventional, word-oriented machines, computers with associative memories are content-addressable. These terms refer to the interrelationship between data as well as to the fact that a memory word or words can be accessed by matching with a selectable field of a special word, rather than by an address. Parallel processing, a related idea and distinct from multiprocessing, allows command sequences to be executed simultaneously over large numbers of data sets.

It was intended to develop a language which incorporates, from the programmer's viewpoint, these facilities in addition to the previously available algebraic, list processing and string manipulating facilities.

For understandable reasons, embedding seemed to be an economical and fairly efficient approach, which also achieves a reasonably high level of machine independence. The presently described version is an extension of SLIP, itself being embedded in FORTRAN IV. (The internal mechanism of SLIP had to be modified to a small extent but the user need not be aware of this fact.) There are only two A[M]PPL subprograms written in assembly lanquage now. Converting often used subroutines and functions, currently coded in FORTRAN, into assembly language would, of course, save considerable machine time with large programs.

Unlike the expensive and inflexible associative memories oresently built in hardware, the size of the Simulated Associative Memory (SaM) in determined by the programmer according to his needs. In fact, he has to specify the ratio between the sizes of SAM and of the Available Space List (AVSL) of SLIP at the beginning of his program. The sum of SAM and AVSL equals the storage area left unused after the compilation of the FOPTPAN program.

The programmer can also build into the program trap points, at which a part of SAM or AVSL is dynamically reassigned to the other part if there is a need for it, i.e. if the size of one memory type decreases below a certain prespecified threshold value. Memory contraction, coupled with a specific form of garbage collection, takes *place* in SAM at this instance.

There is an efficient and fast flow of information between any two of the SAM, AVSL and FOPTrAN memory.

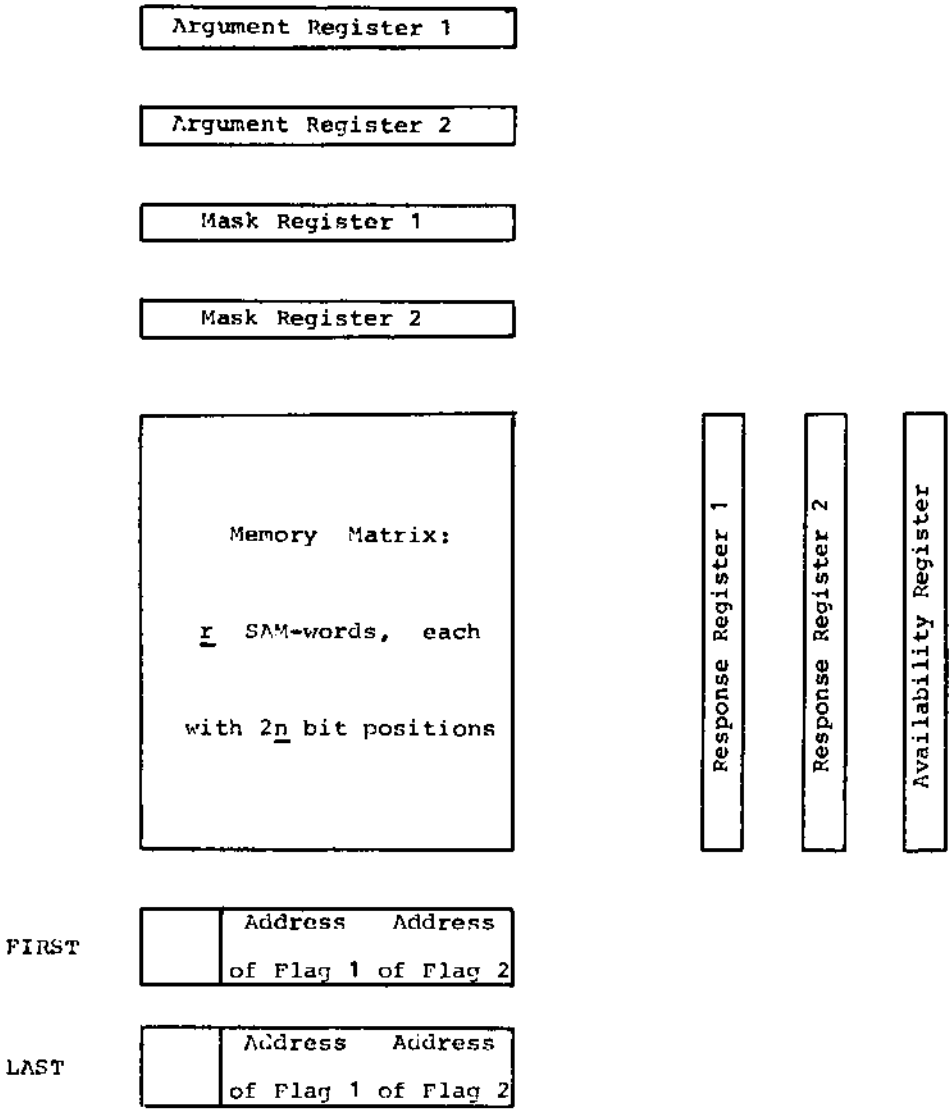
It will be helpful in understanding the organization of AMPPL-II if we consider the diagram in Figure 1.

INSERT FIGURE 1 ABOUT HERE

The main block is the Memory Matrix, which consists of £ SA.M-words*, each having 2n bit positions. There are four special words, the short registers, each n bits long. (On the CDC 6400, there are n=60 bits in a word.) Two of these serve as Argument Registers 1 and 2. The other two represent Mask Registers 1 and 2. There are also three long registers, vertical columns £ bits in length and 1 bit in width. Two ""of these are called response Register 1 and Response Register 2. The third one is the Availability Register. (The role of the words FIRST and LAST in Figure 1 is explained later).

In the following, we shall briefly describe some of the operations used in AMPPL-II and discuss only two, the 'Search and Flag' and 'Set Relation[1], in detail.

♦As will be seen later, every SAM-word consists of two actual memory words.



The Structure of the Simulated Associative Memory and Parallel Processor

FIGURE 1

II. OPERATIONS IN AMPPL-II

(1) Memory Allocation:

As mentioned above, the initial subdivision of free core into SAM and AVSL can be dynamically modified during the execution of the program. Instructions are available to count the number of currently available SAM and AVSL cells, and to accomplish the transfer of free cells from one memory category to the other. At these points, and also at the tine of mass input into SAM (see below), memory contraction and garbage collection in SAM takes place.

(2) Input-Output:

Mass input-output is obtained when FORTRAN arrays or SLIP lists are read into SAM, and when designated SAM-words are loaded into FORTRAN arrays or SLIP lists. Also, wholesale clearing operations can be executed between any two specified S/vM-addresses. (A SAM-address can be conceived of as the index of a one-dimensional array.)

Individual words can be transferred between any two of the SAM, SLIP and FORTRAN memories. Designated parts of SAM can be printed in different modes.

Full words or specified bit configurations can be put into the Argument and Mask Registers, and the contents of the latter can also be read into the FORTRAN memory.

(3) Operations Concerning the Response Registers:

To the AMMPL programmer, SAM appears to contain information in a manner that permits certain basic processes, including reading and writing, to be carried out simultaneously in particular cells. These designated cells contain responding pieces of information and were selected by a previous 'Search and Flag' operation or were deliberately flagged by a special flagging instruction. The •Search and Flag' operations locate and mark SAM-words according to various criteria. Subsequent processes may then be performed on these again and the results can represent Boolean combinations, AND's, OR's and NOT's of consecutive searches.

The basis of comparison is usually* put into one of the Argument Registers. The search is carried out only over those fields of SAM-words that are marked by bits •V in the relevant Mask Register. The success of a search is indicated by flags (bits •!•) at the level of the corresponding SAM-vord in the relevant Response Register. Those SAM-words that have not been used or are no longer necessary are denoted by a tag (bit '1') in the Availability Register.

Two points should be mentioned here. In order to speed un the search processes in SAM, one-directional pointers link flagged SAM-v;ords (see Figure 2) ana, also, end markers indicate the addresses of the SAM-words first and last flagged in the two Response Registers (the contents of the FORTRAN variables FIRST and LAST on Figure 1).

Two other FORTRAN variables, P0INT1 and P0INT2 contain the SAM-addresses of the last SAM-word searched, whether successfully or otherwise, with reference to Response Register 1 and Response Register 2, respectively. (Cf. the possible values of the argument WhICH of the subroutine SLRELS.)

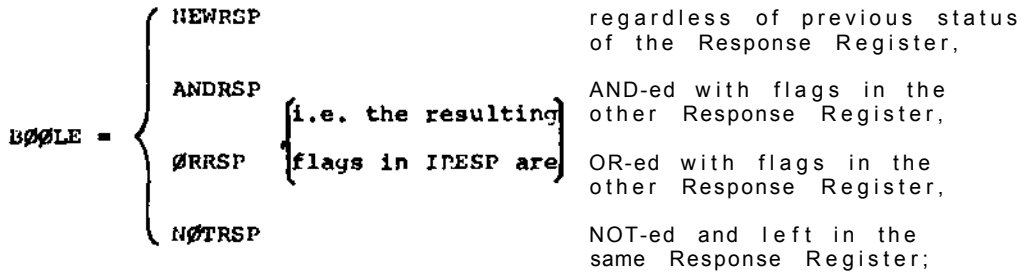
The actual form of the major instruction performing the above described processes is

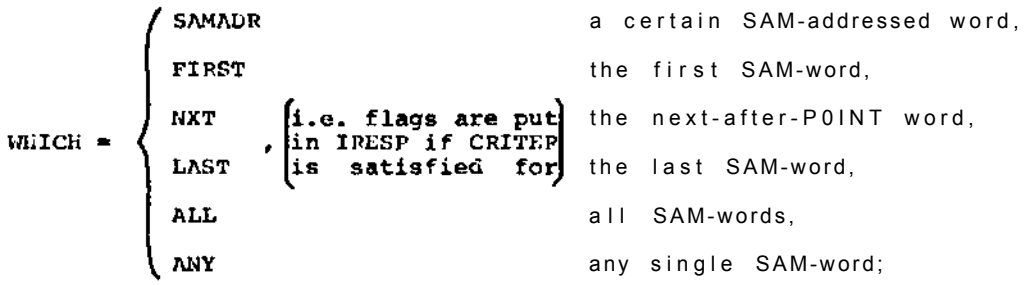
---

If, for example, those numbers are searched for that are greater than a given one. However, if the"criterion of search is, for example, •maximum, the Argument Registers are iqnored.

SERFLG (B00LE, WHICH, CRITER, IARG, IMASK, IRESP)

where





CRITER = {
- NEXTHI — next higher than the one in Argument Register,
- NEXTLØ — next lover than the one in Argument Register,
- NEXT — nearest in absolute value to the one in Argument Register,
- MAX — of largest value,
- MIN — of lowest value,
- GTEQ , {i.e. search for the word(s)} — greater than or equal to the one in Argument Register,
- EQU — equal to the one in Argument Register,
- LTEQ — less than or equal to the one in Argument Register,
- BITSHI — with the hiahest number of bits matched with string in Argument Register,
- BITSLØ — with the lowest number of bits matched with string Argument Register,
- GRPØFM — with the hiahest number of matching groups of M(=2-8) bits regardless of group position, starting from the left;

IARG = { 1 , 2 } i.e. the number of the relevant Argument Register;

IMASK = { 1 , 2 } i.e. the number of the relevant Mask Register;

IRESP = { 1 , 2 } i.e. the number of the relevant Response Register.

A few words of comment are needed here. Two subsequent 'Search and Flag' operations with CRITER=GTEQ and LTEQ yield responsive words of values between given limits. NEXTHI can be performed by two subsequent searches with criteria GTEQ and MIN, similarly **NEXTLØ** is done with criteria LTEQ and MAX. The value of one of NEXTHI and **NEXTLØ,** that is nearer to the value in the Argument Register, yields NEXT. The criteria BITSHI and BITSL0 are useful in comparing non-numerical data and selecting the "most similar- or "least similar" pieces of information, respectively. The number of matching bits can be found as the values of special FORTRAN variables. GRP0FM finds, for example, misprints caused by transposition, missing and added characters. The character set can be represented by groups of 2-8 bits. Since the matching process ignores the position of the groups being matched, there are extra facilities to identify transposition errors. Also, the number of the matching groups is accessible.

There are safeguards to prevent a SAM-word of the wrong information mode (e.g. floating point number instead of alphabetic information) from becoming respondent if its contents happens to be the right bit configuration. A detailed description of this is to be found in 122] .

Finally, it should be noted that there exist instructions to flag and unflag specified SAM-words, to count the number of flagged SAM-words, and to put the SAM-addresses of flagged words in a FORTRAN array or a SLIP list.

(4) Operations Concerning the Availability Register:

As mentioned before, a bit 1 in the Availability Register indicates that the corresponding SAM-word is free. We call this mark a 'tag', as distinct from the •flag' in the Response Registers.

There are instructions which tag and untag SAM-words, count the number of available SAM-words, and which put the SAM-addresses of tagged words in a FORTRAN array or a SLIP list.

(5) Inter-Register Operations:

All the 16 logical operations possible are executable between any two of the short (Argument and Mask) Registers or between any two of the long (Response and Availability) Registers. Here the operands are the bit strings occupying the respective registers.

(6) Processes Regarding Relations:

Besides the 'Search and Flag' operation, these processes are the most significant in AMrPL-II. We shall, therefore, discuss them, also, in some detail.

Let us generalize the concept of an algebraic function and define a Relation (REL) between an Object **(ØBJ)** and a Value (VAL)

REL (ØBJ) = VAL.

Each of the above three entities can be sinqle items or three kinds of lists. The first kind simply contains various equivalent names of the same item. (One can think of synonyms within the given context.) This is called the EQUIVALENT LIST. The second kind of list bears the names of a number of subunits any processing on which is always uniform. An example of these lists nay be the students of a class, who always have the same teacher, always stay in the same classroom, etc. Distinguishing .processes, such as grading of individual exams, are not to be carried out on the elements of so designated lists. Finally, the third kind of list has distinct and, in some respect, independent elements. An example of this could be the pieces of furniture in a certain room if one would like to, say, paint them to different colors.

Also, the programmer can define a Relation as an ordered set of Relations. The combining connectives are:

∧ (and), ∨ (or), ¬ (not),

↓ (concatenated), ← (reverse).

Let us further define a reserved symbol SELF in order to be able to exclude self-referencing in unwanted cases. Finally, the term on the left hand side of a 'concatenated' symbol is considered to be in (Teutonic) genitive. The following examples should make this clear:

(i) PARENT ⇒ FATHER ∨ MOTHER

i.e. a parent is defined a father or mother;

(ii) CHILD ⇒ ← PARENT

i.e. the child is defined as the reverse of the parent;

(iii) GFANDFATIIER ⇒ (FATHERVMPTHER)i FATHER

i.e. the grandfather is defined as the father's or mother's father;

(iv) HUSBAND ⟹ SPOUSE ∧ ¬ WIFE

i.e. the husband is defined as a spouse but (and) not wife;

(v) BROTHER⟹((MOTHER∧FATHER)↓SON∧¬SELF

i.e. the brother is defined as the mother's and father's son but (and) not self; if we wish to include half-brothers as well, we can put

BROTHER ⇒((MOTHER ∨ FATHER) ↓ SON) ∧ ¬SELF

i.e. the mother's or father's son but (and) not self.

As the system reads in these definitions from cards, it checks them in toto for circularity, which it does not accept, and puts them in a Polish prefix form on DEFINITION LISTs. The latter are sublists of the EQUIVALENT LISTs and therefore sub-sublists of the SLIP list CØDEL. (See Figure 2). On the

- -
INSERT FIGURE 2 ABOUT HERE
- -

CØDEL list, the members are either entity (Relation, Object, Value) names, each less than 11 characters long, or the names of the three kinds of sublists mentioned before. The machine address of an item on the CØDEL list is called its Code Number. Whenever a new Relation is defined by the subroutine

SETREL (REL,ØBJ,VAL,K),

the system sets up one (or more) Relation Descriptor word(s) in SAM, which contain(s), in the proper fields, the three code numbers representing the Relation, the Object and the Value. (See Figure 3). K is an integer between 0 and

- -
INSERT FIGURE 3 ABOUT HERE
- -

7, and characterizes each of the three entities whether they represent a single item or non-distinct elements of a list, on one hand, or distinct elements of a list, on the other. In the latter case, as many Relation Descriptor words are established as the total number of combinations possible.

There are altogether seven basic questions a retrieval system for Relations can answer. These are as follows:

(a) Is a particular relation, between a given object and value, true?

(b) What is (are) the value(s) belonging to a given relation-object pair, if any? REL (ØBJ)=?

(c) What is (are) the object(s) belonging to a given relation-value pair, if any? REL(?)=VAL

(d) What is (are) the relation(s) that connect(s) a given object-value pair, if any? ?(ØBJ)=VAL

(e) What relation-object pair(s) belong(s) to a given value, if any? ?(?)=VAL

(f) What relation-value pair(s) belong(s) to a given object, if any? ?(ØBJ)=?

(g) Finally, what object-value pair(s) belong(s) to a given relation, if any? REL(?)=?

The answers are obtainable by using one simple instruction in every case.

Finally, we note two more instructions of considerable power. One of them creates

REL2(VAL) = ØBJ

where the Object and Value are connected by

REL1(ØBJ) = VAL

and

REVREL(REL1) = REL2

i.e. REL1 and REL2 are Reversed Relations. Examples of this are:

REL1	REL2
husband of	wife of
spouse of	spouse of
parent of	child of
loves	is loved by
superset of	subset of
similar to	similar to
greater than	less than
above	below
left to	right to

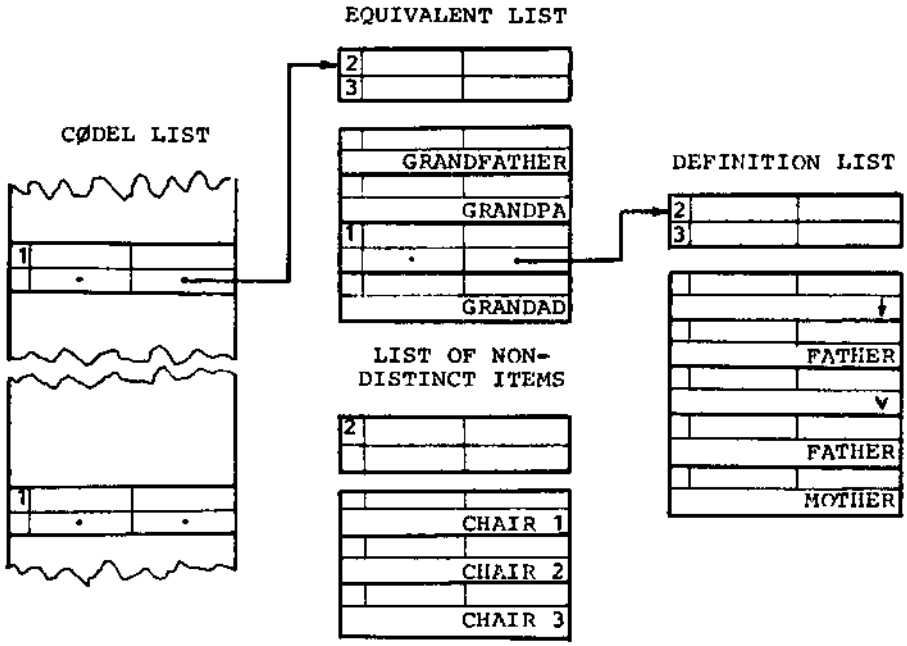
Note that always

REVREL(REVREL(REL1)) = REL1

Another instruction finds X, for which it is true that

A : B = C : X

where A, B, and C are any of the three entities, Relation, Object, or Value; A, and C, on one hand, and B and X, on the other, are of the same type, and the third entity in the Relation Descriptor words containing A, B and C, X is the same for both. (Occurrences of all possible combinations of A and B are considered.) Two examples should make this clear:



An Exemplary Segment of the

CØDEL List and Its Sublists

FIGURE 2

CODE NUMBER 1	CODE NUMBER 2	CODE NUMBER 3
1 1	ADDRESS OF DOWNWARDS NEAREST FLAG 1	ADDRESS OF DOWNWARDS NEAREST FLAG 2

1 2 3 4 5 6

SAM-word as Relation Descriptor

FIGURE 3

(a) Suppose in Sam, we have Relation Descriptor words standing for

```
MOTHEP          OF (JEANNE) = FRENCH
MOTHER TONGUE   OF (JOSÉ)   = SPANISH
```

If

A = JEANNE,
B = FRENCH,
C = JOSÉ

the resulting X will be SPANISH since Jeanne's relation to French is the same as that of Jose's to Spanish — these are the mother tongues of the people in question.

(b) Let SAM now contain

```
UNCLES OF (JACK) =  JOE, BILL, PETER
AUNTS OF (JACK)  =  MARY, CARON
```

If

A = UNCLES OF
B = JOE, BILL, PETER
C = AUNTS OF

the resulting X will be the list with MARY, CARON since JOE, BILL, PETER are the uncles of the same person whose aunts are MARY, CARON .

(7) Parallel Operations over SAM;

Here, we briefly list some basic but high-level instructions that should be useful both in numerical and non-numerical applications 2

A constant word in one of the Argument Registers can be added to, subtracted from, multiplied by, divided into. Boolean AND-ed and OR-ed with designated SAM-words through one of the Mask Registers,

The above operations can also be performed between any two fields, and the Boolean NOT of any single field, of designated SAM-words.

Specified fields of designated SAM-words can be cleared.

Designated SAM-words can be shifted to the left or to the right by specified number of places.

Single elements, vectors, planes, cofactors or all elements of an array can be flagged if the array was read into SAM according to the usual mapping order.

Vector addition, subtraction and scalar multiplication are single instructions.

Also, single instructions yield the determinant and the inverse of one, and the product of two, matrices.

III. AN OVERVIEW

We have tried to give a short outline of a new computer language. We, however, feel it is more than "just another language" — it represents another philosophy of, another approach to, problem solving. After all, it is only the sequential design of the von Neumann-type machines that has imposed upon the computing community the presently prevalent but often quite unnatural computational methods. Even using these conventional methods, AJ'PPL-II (a) should decrease the length of written programs and (b) should simplify the writing, debugging and understanding of programs. (It has very powerful diagnostic facilities.) There is, however, a significant trend, as can be seen in the referenced literature, to develop new algoritlims and techniques that make use of content addressability and parallel processing, expose latent parallelism, and introduce computational redundancy. We hope AMPPL-II will enhance this trend.

We have had only limited programming experience with the language. The following, as yet incomplete, projects are representative examples:

(1) a query system, which can be continually updated, dealing with complex kinship structures;

(2) simulation of a learning and self-adapting organism in a hostile environment;

(3) empirical proofs of conjectures in computational linguistics;

(4) simulation of a demographical problem; and

(5) scheduling classrooms, teachers and students.

We have found that, although one must pay a certain price in machine time and available memory, the programming ease achieved is quite significant. (The disk resident system consists of roughly 8k

words for each SLIP and AMPPL-II. Only the needed subprograms are called into core.) In the INTRODUCTION, we listed a number of broad areas of application in which AMPPL-II should prove useful. Now we can be more specific in terms of problem characteristics. We expect to save considerable programming effort in using the language whenever

(i) data are to be addressed by a combination of various sets of reference properties,

(ii) data elements satisfying the above reference properties are scattered throughout the memory in a sparse and random manner,

(iii) data elements dynamically change their location in the memory as a consequence of the information processes acting on them,

(iv) identical sequences of processes manipulate on distinct, non-interacting data elements,

(v) the ratio between concurrently and serially executable processes is reasonably high.

These criteria of language applicability occur to some extent with practically every complex programming problem. The availability of a conventional algebraic language with the AMMPL-cum-SLIP package renders programming more efficient.

We intend to study in a later paper the numerous issues involved in using AMMPL-II in various fields. Here, we only wish to point to the fact that the proposed system, to an extent, is capable of simulating two distinct kinds of associative memory. In the exact associative memory, the information processes are performed on the basis of finding the intersection of several matching descriptors. Because of the non-uniqueness of many associations and, also, because retrieval requests may be incomplete, there can be several respondent pieces of information. However, ill-formulated and imprecise tasks cannot, in general, be solved, and there is no logical "interpolation" or "extrapolation".

On the other hand, in a non-exact associative memory these restrictions do riot apply. Associations connect statistically related entities, too. The measure of "nearness" is an important concept. Various counting processes, the criteria of search for the most and least similar items (BITSIII and BITSL0) and matching groups of bits (GRP0FM) represent steps in this direction. Biological systems, of course, incorporate both of the above described associative memories.

REFERENCES

[I] Findler, N. V., On a Computer Language which Simulates Associative Memory and Parallel Processing (Proc. NATO Symposium on Comp. Systems, Lyngby, Denmark, 1967)

[2] Findler, N. V., Towards Making Machines More Intelligent (Proc. Fifth International Congress on Cybernetics , Namur, Belgium, 1967)

[3] Feldman, J. A., Aspects of Associative Processing (M.I.T. Lincoln Laboratory Technical Note 1965-13, April 196S)

[4] Rovner, P. D. and J. A. Feldman, An Associative Processing System for Conventional Digital Computers (ibid, 1967-19, April 1967)

[5] Rovner, P. D. and J. A. Feldman, The LEAD Language and Data Structure (Proc. IFIP Congress 68, preprints, pp. C73-C77, 1968

[6] Dodd, G. G., R. C. Beach, and L. Rossol, APL - Associative Programming Language User's Manual (General Motors Research Laboratories Publication GMR 622, July 1967)

[7] Hanlon, A. G., Content-Addressable and Associative Memory Systems. A Survey (IEEL Trans, on El. Comp., EC-15, pp. 509-521, 1966)

[8] Lehman, M., A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors (Proc. IEEE, 54, pp. 1889-1901, 1966)

[9] Ewing, P. G. and P. M. Davies, An Associative Processor (Proc. 1964 FJCC, pp. 147-158, 1964; Spartan: Baltimore)

[10] Fuller, R. H. and J. M. Salzer, Associative Processor Study (General Precision, Inc. Peport; DDC No. AD-608427, T95TH

[II] Fuller, R. II., J. C. Tu, and R. M. Bird, A Woven Plated-Wire Associative Memory (Proc. Nat. Aerospace Electronics Convention 1965)

[12] McKeever, E. T., The Associative Memory Structure (Proc. 1965 FJCC, pp. 371-388, 1965; Spartan: Baltimore)

[13] Slotnick, D. L. (Chairman), Special Session on Parallel and Concurrent Computer Systems (Proc. ITIP Congress 65, pp. 319-322, 1965; Spartan: Baltimore)

[14] Chu, y., A Destructive Readout Associative Memory (IEEE Trans. on El. Comp., EC-14, pp. 600-605, 1965)

[15] Hasbrouck, B., N. s. Prywes, D. Lefkovitz, and N. Kornfield, Associative Memory Computer System - Description and Selected Naval Applications (Computer Control and Command Co. Report No. 25-101-11, 1955)

[16] Fuller, R. H., R. M. Bird, and R. M. Worthy, Study of Associative Memory Techniques (General Precision, Inc. Report; DDC No. AD-6!H51£, 1955)

[17] Dugan, J. A., R. S. Green, J. Minker, and W. E. Shindle, A Study of the Utility of Associative Memory Processors (Proc. ACM Nat. Meeting, pp. 347-360, 1966; Thompson: Washington)

[18] Prywes, LI. S., Man-Computer Problem Solving with Multilist (Proc. IKEK, 54, pp. 1788-1801, 1966)

[19] Knapp, M. A., R. H. Fuller, R. M. Bird, J. L. Cass, and J. M. Salzer, Papers presented at the ONR/RADC Seminar on Associative Processing (Mimeographed Proceedings, Washington, 1967)

[20] Stone, H. S., Associative Processing for General Purpose Computers through the Use of Modified Memories (Proc. 1968 FJCC, pp. 949-955, 1968; Thompson: Washington)

[21] Weizenbaum, J. Symmetric List Processor (Comm., ACM, 6, pp. 524-544, 1963)

[22] Findler N. v., User's Manual for the Associative Memory, Parallel Processing Language, AMPPL-II in press; SUNYB Computing Center Press)