INTRODUCTION

Engineering design and computer-aided problem solving occur in an atmosphere in which the relationships between the data elements are utilized. For example, information retrieval, computer-aided drawing, electrical network design, and engineering design systems are among those whose operation depends on efficient data manipulation and association techniques.

Problem solving with a computer should give a user an opportunity to continually change and restructure his data. In effect, he should be able to reach into his data structure and ask questions about the data: What pieces of data is this one related to? What is affected if the numeric value attached to the data element is altered? Likewise, he should be able to specify operations to be applied to a group of elements: Collect these things together! To that collection of data elements apply algorithm XXX! Create a new element and put it into a collection!

Algebraic programming languages permit elaborate mathematical expressions to be stated in a clear and direct manner. The user who is uninterested in assembly language techniques can state a procedure for arriving at a solution to a problem, in a high-level language.

The solution of many classes of problems, however, requires data handling and here the existing languages break down. The algebraic languages do not permit the user to treat his data in the same clear and direct manner as he treats mathematical formulae. He has to become an expert in two fields—the one, his specialty and the other, data handling.

APL was conceived at the General Motors Research Laboratories to satisfy the need for convenient data association and data handling techniques in a high-level language. Standing for ASSOCIATIVE PROGRAMMING LANGUAGE, it is designed to be embedded in PL/I as an aid to the user dealing with data structures in which associations are expressed.

The language offers facilities for establishing relations between the data elements, developing and restructuring the data structure as the program proceeds, and manipulating data without programmer concern about the complicated mechanics used to establish the data relationships. In addition to a full PL/I capability, the language provides

a. Symbolic data references.
b. Use of English verbs and phrases in data manipulation statements.
c. Hierarchy of constituent parts permitting orderly formulation of data relationships.
d. N-dimensional data associations.
e. Automatic extension of addressable memory beyond the confines of high-speed core.
Machine independent implementation is possible by writing the APL preprocessor in PL/I. However, the deferred and incomplete pointer features of PL/I have made this impossible at the moment.

APL contains many of the facilities underlying such list processors as IPL, COMIT, LISP, SLIP, but it gives the user a more complete data manipulation capability with less effort. Unlimited block sizes accommodate any PL/I structure and the data linking techniques offer a full class of forward, reverse, and n-dimensional associations between the data structures.

ELEMENTS OF AN ASSOCIATIVE DATA STRUCTURE

The associative data structure underlying APL can best be described in terms of the hierarchy of its elements. The structure is the framework in which the data items reside. Each structural element is tied to other elements by means of their relationship, and can have other elements tied to it. In addition, each structural element is described by data pertaining only to that member.

The basic element of an associative data structure is an ENTITY. Any identifiable type of element in a data structure such as a job, a surface, or an automobile is treated as an entity. Entities are described by ATTRIBUTES (more precisely, the values of attributes). For example, an entity which is a Job can have the attributes Job ID, Job Title, Duration, Finish Date, and Start Data.

Many different types of entities may exist in a data structure. An engine, frame, tire, windshield are all types of entities appearing in the data structure of an automobile. In addition, many copies or instances of each entity type may abound.

Related entities may be grouped into a SET. This SET can be: (1) owned by an entity; or (2) it can be resident in data space and referenced independently of any entity. An example of the former is the collection of jobs which must be completed before a given job is started: here the collection is tied to an entity and is called a SUBSET of the entity. A set of type (2) might be the set of all jobs in the file. This set is not attached to any entity but is known to all programs using the file.

An entity is depicted as a contiguous block of addressable memory having one or more of the following:

a. References to the subsets belonging to the entity.

b. References to the one or more sets to which the entity may belong. These are known as associative set reference links.

c. Data attributes associated with the entity.

The implementation technique used stores pointers in the entity block to the subsets of the entity and to the sets to which the entity belongs. In Fig. 1, this is portrayed with the X followed by an arrow indicating the subset links and the double-ended arrows indicating the associative reference links.

Entities are linked into a set by means of Roberts’ ring structure.* This structure was chosen because it permits rapid movement through a set in one direction. At the same time it permits movement through a set in a backwards fashion or to the entity to which the set is tied with only a small increase in overhead. Because of these properties, random addition and deletion of entities in a set is easily accomplished.

Figure 2 shows entities \{B,C,D\} connected to a set belonging to A. This is expressed as \{B,C,D\} \epsilon A.

An entity can have many subsets belonging to it and in turn can belong to an indefinite number of sets. The contiguous entity structure permits subsets and associative references to be found directly in the entity rather than moving along a list.

Figure 3 shows D and E belonging to a subset of B. E is also a member of a subset of C which is a member of a subset of A. These relationships can be logically expressed as \{D,E\} \epsilon B, E \epsilon C, and C \epsilon A.

The data attributes describing the entities may assume one of two forms, direct or associative. A

* Copies of a paper describing this structure can be obtained from Dr. Lawrence Roberts, Lincoln Laboratory, Lexington, Mass.
A direct attribute is a PL/I data structure residing in an entity block and addressable as part of the entity. Associative attributes are used to express relations between entities or between an entity and a set of which it is a member. Direct attributes are illustrated by the geometric coordinates which describe a point. However, the distance between that point and some other point is an attribute value associated with both points; both points as well as the name of the attribute must be known to retrieve its value.

In some cases an entity is a conditional member of a set. Here, one or more attributes are needed to fully qualify the membership. In Fig. 4 the line is a member of the boundary line set for surface 1 between the values $X_1$ and $X_3$ and a member of the boundary line set of surface 2 between the values of $X_2$ and $X_4$. These associative attributes may be specified and retrieved within APL.

**PROGRAMMER TOOLS**

Now that the elements of an associative data structure have been described, let us turn our attention to how it is used. The tools for the data manipulation can best be described in terms of

a. Data set declarations,
b. Attribute set and entity references,
c. Data structure manipulation statements.

**Data Set Declaration**

The entities, sets, and attributes making up an associative data structure are specified in a series of statements preceding the programs to be translated and compiled. This procedure insures compatibility across jobs and eliminates repetitive declarations in the programs compiled within the same job environment. Sufficient information is provided for the opening of files or initialization need prior to the execution of the first instruction in the program.

The specification of an entity includes the entity type, i.e. an alphanumeric designation by which that type of entity will be referenced, the names of the subsets of the entity, the names of the associative sets linking the entity and the names of the direct and associative attributes of the entity.

A separate declaration of each set is also required. In these statements the procedure for the storage of entities within the set is specified. The storage procedures can be overridden by a specification from the programmer any time in his program. (Figure 8, appearing later, shows the declaration of a JOB entity having a PRED subset, associative links to the PRED and ALLJOB sets, and a number of direct attributes.)

Associative attribute declarations are prefaced with an ATTRIBUTES* identifier. A point entity having direct attributes $x, y, z$ and an associative attribute $distance$ is declared by

```
DECLARE 1 point ENTITY,
  
  2 distance ATTRIBUTES* float,
  2 x ATTRIBUTE float,
  2 y ATTRIBUTE float,
  2 z ATTRIBUTE float;
```

**Figure 2. Set of entities.**

**Figure 3. Multiple sets of entities.**
Attribute, Set, and Entity References

Many instances of each type of entity can appear in a data structure and it is often necessary to reference several of these instances simultaneously. To do this a variable of data type ENTITY has been introduced in APL. At the time an entity is created or obtained through a referencing mechanism, an entity variable is set to identify the correct instance. Figure 5 shows five entities; one being referenced by entity variable A, one by entity variable B, and three un-referenced.

Sets are referenced through the entity instances to which they belong by means of the notation

\[
\langle \text{entity instance} \rangle, \langle \text{set name} \rangle
\]

In the figure, the fully qualified name of the PROP subset is given by A.PROP.

The value of a direct attribute is referenced by qualifying its name with the names of the relevant sets and entities. The names are separated by a period "." in a manner similar to the period qualifier used in PL/I data structure names. Accompanying each set name is an integer specifying the ordinal number of the desired entity. For example, if A currently references an entity whose PROP subset has a third entity whose X value is desired, the latter is referenced by A.PROP(3).X (see Fig. 5). The Z value of A is obtained by A.Z.

In another example, the X's of the first three entities in the subset PROP are added together and placed in the first Y by the assignment statement

\[
A.\text{PROP}(1).Y = A.\text{PROP}(1).X + A.\text{PROP}(2).X + A.\text{PROP}(3).X
\]

The attribute expression is usable in more than one level. The attribute COST of entity B may be designated by A.PROP(3).Q3(1).COST. The last entities in a set may be referenced by negative integers. Thus, A.PROP(-1).X is the last X in the subset and A.PROP(-2).X is next to the last X.

Associative attributes are referenced by the format

\[
[\text{entity - 1}, \text{entity - 2}].\text{attribute}
\]

or

\[
[\text{entity, set}].\text{attribute}
\]

depending on whether the associative attribute is between entities or between an entity and a set. If the attribute distance existed between entities E1 and E2, its value could be specified by

\[
[E1, E2].\text{DISTANCE} = 5;
\]

Both direct and associative attribute references may be used in any APL or PL/I statement where a value is to be retrieved or assigned.

Data Structure Manipulation Statements

The statements provided by APL for data manipulation have a key word syntax. The statements complement existing PL/I statements and may be used anywhere within the source program. The six statements and their meaning are:

a. CREATE entity-type CALLED entity-variable;

Space for an entity of the specified type is allocated in the users file and the designated entity variable is set to reference the new entity.

b. INSERT entity-variable IN set;

The designated entity is made a member of the designated set.

c. REMOVE entity-variable FROM set;

The entity is either removed from the designated set or from all sets of which it is a member.

d. DELETE \{ entity-variable \} ;

The first option deletes the specified entity from the file and its space is returned to the free-space list for reallocation. The second option removes all en-

![Figure 4. Example of associative attribute.](image)

![Figure 5. Attribute references.](image)
entities from the specified set and deletes them. The delete routine is recursive in that the entities belonging to subsets of the given entity will also be deleted if they belong to no other subset.

c. FIND entity-variable = entity specification 
    [,WITH β] [,UNTIL β] [,ELSE statement];

In its basic form the FIND statement sets an entity variable to reference an entity whose identity is given by the clauses of the statement.

The entity specification can assume one of two forms; the specification of an entity which is a member of a set or the specification of an entity whose subset contains a designated entity. These can be illustrated by

FIND e1 = (3) line ⊆ e2.boundary;

which finds the third LINE entity contained in the BOUNDARY set of E2 and

FIND e2 = boundary⊃e1;

which finds the entity whose BOUNDARY SET contains E1. Because of the lack of characters in the PL/I 60 character set, the word IN is substituted for ⊆ and CONTAINS is substituted for ⊃ in the APL implementation.

The WITH phrase contains a Boolean expression to be applied to each entity in the set. If an entity satisfies the expression, a count is increased until the correct entity is found. The third POINT entity having an X value of 3 would be specified by

FIND pt = (3) point ⊆ el.ptset, WITH
    x = 3;

The Boolean expression in the UNTIL phrase designates the condition which terminates the search. Should the search be terminated either by exhausting the set or by action of the UNTIL phrase the ELSE clause is executed. The WITH, UNTIL and ELSE phrases are all optional.

Unless otherwise specified the statement is executed on a set in the order in which entities are inserted in the set. The search can be made in a backwards direction through the set (specification of negative indices) and can be directed to start at some element in the set (by inserting ⊆ FROM entity variable ⊃ in the entity specification.

d. FOR EACH entity-variable = entity-
    specification, [,WITH β] 
    [,UNTIL β];

Simply put, this statement is a FIND statement in a DO loop. The block of statements between the FOR EACH and END are executed for each entity variable meeting the specifications.

The APL manipulation statements also augment the Boolean capabilities of PL/I with two additional tests: (1) test a set for emptiness; and (2) test an entity for membership in a designated set.

CONTROL OF EXTERNAL STORAGE

APL operates on the philosophy that external storage should merely be treated as a larger addressable memory. Therefore, special PUT and GET, READ and WRITE statements are not necessary. The system operates in conjunction with a software data paging supervisor. This supervisor intercepts all data references and supplies the program with the correct data items.

At program initialization time a buffer is established in core where, upon request, the necessary data pages are placed. Each data page contains a number of entities. The address of an entity specifies both the page within which the entity resides and the physical location of the entity within the page.

Upon intercept of a data request the supervisor examines its stock of in-memory data pages. If the correct page is present, the address of the correct entity or set in the page is returned to the program. If the data item does not reside in the core buffer, an unused or old page is transferred out and the correct page is found in the external storage and read into the paging buffer. The relocated address of the requested set or entity is then returned.

The system can also operate in a non-paging mode. Here, addresses of all data items are absolute and no paging or pointer interception occurs. Limitations of the non-paging mode are the confines of core memory for a work space and the external file manipulations required if the user wishes to save his data set.

Operation in a time sharing and paging environment is also being anticipated. At this time the paging supervisor will be lifted out and replaced by the hardware paging and segmentation functions of the system.

Control of External Storage

APL operates on the philosophy that external storage should merely be treated as a larger addressable memory. Therefore, special PUT and GET, READ and WRITE statements are not necessary. The system operates in conjunction with a software data paging supervisor. This supervisor intercepts all data references and supplies the program with the correct data items.

At program initialization time a buffer is established in core where, upon request, the necessary data pages are placed. Each data page contains a number of entities. The address of an entity specifies both the page within which the entity resides and the physical location of the entity within the page.

Upon intercept of a data request the supervisor examines its stock of in-memory data pages. If the correct page is present, the address of the correct entity or set in the page is returned to the program. If the data item does not reside in the core buffer, an unused or old page is transferred out and the correct page is found in the external storage and read into the paging buffer. The relocated address of the requested set or entity is then returned.

The system can also operate in a non-paging mode. Here, addresses of all data items are absolute and no paging or pointer interception occurs. Limitations of the non-paging mode are the confines of core memory for a work space and the external file manipulations required if the user wishes to save his data set.

Operation in a time sharing and paging environment is also being anticipated. At this time the paging supervisor will be lifted out and replaced by the hardware paging and segmentation functions of the system.

The software paging mechanism forces limitations on the entity size for efficient processing. The entity size must be less than the page size. This restriction will be removed when hardware paging and segmentation is implemented.
Example

Project scheduling programs offer some interesting problems which can be readily solved by programming in APL.

Figure 6 is a simplified project schedule. This schedule shows the activities which must occur in moving the project from the MASTER PLAN to the end of TEST PRODUCT. The activities and their titles are represented by the solid lines in the figure. The dotted lines show the precedence relationship between the activities in the project. For example, MASTER PLAN has no predecessors and can begin immediately. The activity PRODUCE E has the immediate predecessors PLAN B and PLAN C and cannot be started until the predecessor jobs have been completed. The predecessors, therefore, determine the ordering of the activities.

Each activity also has a duration. This is a measure of the time required to complete the task once it has been started. In the example, MASTER PLAN, PLAN B and PLAN C have respectively 4, 7 and 13 days duration. PRODUCE E has 13 days duration but cannot be started until both PLAN B and PLAN C have been completed. This pushes its start date back to the fourteenth day.

An APL program will be written to read in descriptions of activities, establish the required data structure and compute the start and finish dates for each activity.

The input to the program is information about each job (Figure 7). This consists of an ID number and title by which the job will be identified, the duration of the job, and a listing of the immediate predecessors of the job.

The data structure used in this program has one type of entity called JOB. Each JOB entity has a subset PRED which is the collection of all immediate predecessors to the JOB. A JOB may be a predecessor of another JOB; if so the association is through the PRED associative set member link. These links are internally expanded so that JOB can be a predecessor of any number of other JOB entities. All JOBS are collected together in the set ALLJOB by means of a second associative reference link.

Each entity has the attributes duration and title given in the job information form. In addition, it has a start and finish date which will be calculated from the location of the entity in the precedence chart.

Figure 8 is the APL declaration of a JOB entity, the entity attributes and the sets relating the entities as outlined above.

The specification further outlines that PRED sets are a first-in-first-out assignment (FIFO) and that the ALLJOB set is ordered on the ID of its constituent entities.

```
/*DECLARE THE ENTITY CALLED JOB*/
DECLARE 1 job ENTITY, 2 pred SET fifo, 4 pred MEMBER, 7 title CHAR (20), 9 duration FIXED, 11 id FIXED;
DECLARE 1 alljob MEMBER, 13 sw BIT (1), 15 alljob SET ORDERED INCR ON id;
```

Figure 8.

Figure 9 illustrates a part of the data structure to be constructed by an APL program operating on the declarations given in Fig. 8. In the structure JOB C is expanded to show the attribute and set relationships. JOBS A and B are members of the

---

**Figure 7. Job information form.**

<table>
<thead>
<tr>
<th>ID</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DURATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREDECESSOR (1)</td>
</tr>
<tr>
<td>PREDECESSOR (2)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

---

**Figure 6. Project schedule.**

---

<table>
<thead>
<tr>
<th>MASTER PLAN</th>
<th>PLAN TEST</th>
<th>TEST PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>17</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLAN B</th>
<th>PRODUCE D</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLAN C</th>
<th>PRODUCE E</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>
Figure 9. Data structure for scheduling program.

PREDecessor set of C and JOB C is a member of the PRED set of D. In addition C has the attributes shown.

Figure 10 shows the APL program. This program, plus the declaration statements given in Fig. 8, are what are given to the APL processor.

The program is broken into several parts:

Statements 2–4 —declarations.

Statement 5 —create a file known as SCHJOB. All future CRE­

ATEs are assumed to be in this file.

Statement 6 —create a subset link in the file for ALLJOB.

Statements 7–10 —read in the ID of a new JOB and create the JOB if this has not already been done.

Statements 11–15 —place the predecessors of the new JOB in its PRED set. If a predecessor has not been defined, create a JOB whose duration, title and predecessors are to be supplied later.

Statements 16–23 —internal procedure that creates a JOB entity, sets its ID and SW. SW is a switch used in statements 25–28 to establish the finish dates of the activities.

Statement 24 —read in start DATE.

Statements 25–28 —find a JOB whose prede­
cessors have been resolved or which has no prede­
cessors.

Statements 29–36 —assign the JOB a start date.

Statements 37–43 —repeat for all JOBS; when done check for unassigned JOBS (these have cycles caused by inconsistent data) and print a com­

ment.

SCHEDULE:PROCEDURE OPTIONS (MAIN);
DECLARE (y,z) ENTITY,
(p.date) FIXED,
test BIT(1);
CALL CRFILE ('schjob');
CREATE alljob;
READ:GET LIST (ident);
IF ident = (-2) THEN GO TO LINK;/* -2 SIG-
NALS END OF INPUT*/
FIND z = (1) job IN alljob, WITH z.id = ident,
ELSE CALL DEFINE (ident,z);
GET LIST (z.title,z.duration);
MORE:GET LIST (p);
IF p = (-1) THEN GO TO READ; /* -1 12
MARKS END OF PREDECESSOR LIST*/
FIND y = (1) job IN alljob, WITH y.id = p,
ELSE CALL DEFINE(p,y);
INSERT y IN z.pred;
GO TO MORE;
DEFINE:PROCEDURE (arg1,arg2);
DECLARE arg1 FIXED,
arg2 ENTITY;
CREATE job CALLED arg2;
arg2.id = arg1;
arg2.sw = 'O'b;
INSERT arg2 IN alljob;
RETURN;
END DEFINE;
LINK:GET LIST (date);
AGAIN:
test = '0' b;
END2:FOR EACH z = job IN alljob,
WITH z.sw = 'O'b;
FIND y = (1)job IN z.pred, WITH y.sw = 'O'b,
ELSE GO TO GO1;
GO TO END3;
GO1:z.sw = '1'b;
IF z.pred = EMPTY THEN
z.start = date;
ELSE DO
z.start = MAXVAL(finish,z.pred) + 1;
z.finish = z.start + z.duration;
test = '1'b;
END3:END END2;
IF test THEN GO TO AGAIN;
FIND z = (1)job IN alljob, WITH z.sw = '0'b,
ELSE DO; PUT LIST ('satisfactory run');
RETURN;
END;
PUT LIST ('incorrect data');
END SCHEDULE;

Observe that some of the statements, such as CREATE and FIND, require the variable Y or Z to be the representative of each entity in the set. This is an entity variable described earlier and its declaration is in Statement 2.
At times it is convenient to designate an entity as being the current entity for purposes of attribute identification. In these cases unqualified attribute references will be qualified by the current entity. For example, if the declaration is

DCL Y ENTITY, Z ENTITY CURRENT (JOB) . . .

then Statement 34 could be written

FINISH = START + DURATION;

with the implication that this operation is to be done in the current job entity.

This is but one example of APL effectiveness. An extended version of this problem which organized the activities and printed out a table was written some time ago in NOMAD.13 The writing of the program took three weeks and required 500 statements. It is estimated that a 3 to 1 savings in programming time and a 5 to 1 savings in number of statements would have been realized, had the program been written in APL.

SUMMARY

The writing of programs in which the relation and dynamic manipulation of data elements is important can be made much easier with APL, a language designed at the General Motors Research Laboratories to fulfill this need. It closes the gap between algebraic languages and data handling needs with statements and defining capability permitting the description of data relationships. It further provides a means for efficiently expanding the programming environment with automatic file handling facilities.

APL statements are embedded in PL/I and are sifted out by a preprocessor. Thus, the user has a full PL/I capability in addition to the APL data handling statements. Output from the preprocessor is PL/I statements and subroutine calls.

Two new data types are introduced in APL, the SET and the ENTITY. These have a semantic relationship with the existing data types. This relationship is further clarified by the use of SET and ENTITY variables in the APL statements and control phrases.

The APL preprocessor is not a simple macro expander. The new data types, the entity declarations, the expanded use of the "." data qualifier and the expanded Boolean tests call for a processor. The processor tabulates information from the declarations and data types and later references the tables to produce an efficient symbolic PL/I output.

One goal of a high level language is to make computerized problem solving easier. By using APL commands the programmer is able to work with the data in terms of function and not in terms of the petty details and the pointer manipulations which actually perform the task.

ACKNOWLEDGMENTS

The design of APL was made possible only through the efforts of many members of the staff at the General Motors Research Laboratories. One of the first evaluations of APL was made by John T. Murray whose subsequent assistance in defining the language merits specific mention.

REFERENCES