Advanced applications of APL: logic programming, neural networks, and hypertext

by M. Alfonseca

This paper reviews the work of the author on the application of the APL and APL2 programming languages to logic programming, emulation of neural networks, and the programming of hypertext applications.

The last decade has witnessed the emergence and maturation of a whole set of new fields and techniques in computer science, such as logic programming (which actually started in the 1970s), neural networks, object-oriented programming, genetic algorithms, and a few others. APL (and its successor APL2) remains abreast of the times as a programming language and has demonstrated its capability for all of these exciting new fields.

This paper summarizes the previous work of the author in three of the indicated fields. The first section, on logic programming, describes the design of a logic programming auxiliary processor, capable of performing declarative logic inferences similar to Prolog, that can be invoked and used from an APL workspace. This processor is now a part of the APL2/PC IBM product.

The second section, on neural networks, describes how APL can be used to model, teach, and implement these modern structures which, though descending directly from the perceptrons of the 1960s, have now revived with a new strength and are being applied to new, interesting fields.

Finally, a third section summarizes why APL2 is extremely apt for the development of object-oriented applications and describes in some detail a hypertext application built on these lines.

APL and logic programming

The literature on APL shows that there has been a long-standing discussion about the usefulness of this language for artificial intelligence applications. This usefulness is considered a direct consequence of the great power of the language, the ease of programming with high-order data structures, or the possibility of using a "parallel" approach to solve certain problems. Reference 1 gives more details on the latter.

In particular, the new list structures introduced in the APL2 form of the language[2] provide APL with all of the power of LISP, the classical language for artificial intelligence.[3,4]

Several attempts have been made to build expert systems using only the current power of the language, either with APL or APL2.[5–9] Building an expert system usually requires the implementation of

Table 1 Reserved symbols for AP998

APL Symbol	Standard Symbol	Meaning
←	<-	IF
∧	&	AND
∨	|	OR
~	-	NOT

an inference machine, or some version of logic programming, in APL. The problem here is speed. However, some of the approaches find highly original ways to solve this problem.

A related approach is the emulation of Prolog-like rule-based inferences. Although such emulation has been done more than once,[10,11] this approach is usually too slow, for it boils down to interpreting an interpreter.

A better solution to this problem would be the implementation of a Prolog-like inference processor in a lower-level language in such a way as to be easily accessible from normal APL programs. In this way, powerful hybrid systems could be implemented. Applications built in APL using this "logic auxiliary processor" would gain access to a whole class of new possibilities (logic inferences, "natural-like" language, nonprocedural programming) while at the same time maintaining all of the APL numeric calculation and symbolic manipulation capabilities.

This inference processor is already written and is a part of the APL2/PC product. It is an auxiliary processor, called AP998, accessible from APL2 in the usual way through shared variables, and incorporates a subset of a Prolog-like interpreter.

It has been said that this method is not really an APL solution, since it does not use pure APL programs but instead adds one external program (the auxiliary processor) written in a different language. I think this criticism is unfair, because:

- Auxiliary processors are, and have been for a long time, a part of APL. The fact that they are included in the products proves this assertion.
- APL allows the construction of auxiliary processors in different languages, and this capability is a plus, not a minus, of the language. It is a well-known fact that APL as an interpretive language has a certain impact on performance. The standard solution (avoiding loops in the code) is not always feasible, especially when cascaded results are involved, that is, those processes where the next value to be computed depends on previously computed values. In those cases, it is a great advantage to be able to speed the system up by programming the bottlenecks in a lower-level language. If this can be done in such a way that the resulting auxiliary processor is of general application and can be reused in very different contexts, APL becomes richer and increases its power for future applications.

The remainder of this section describes the logic inference auxiliary processor, AP998.

The logic language. The logic language implemented by AP998 is a subset of Prolog using only infix notation. The lexical elements of the language are the following:

- Words—A word can be defined as any character string not including spaces. Uppercase and lowercase are considered to be equivalent. Examples are:

```
JOHN
IS-FATHER-OF
25
```

- Reserved symbols—Certain symbols have special meaning for AP998 and should not be used outside their context. To be recognized, these reserved symbols must be separated from adjacent words by at least one space. Each meaning can be represented by two different symbols, one of which is easier to represent with the APL keyboard, whereas the other is easier to represent with the standard keyboard. The symbols are shown in Table 1.
- Synonyms—Certain words can be defined as synonyms for the reserved symbols. In this way, many natural languages are recognized by AP998. In English, the synonyms recommended for the symbols are the words indicated in Table 1 under the heading Meaning. Only one synonym may be defined for each meaning at a given time.
- Variables—Any character string starting with the "star" symbol (the asterisk, *) represents a variable. Examples are:

```
*X
*CASE
*1
*
```

The syntactic elements of the language are the following:

- Clauses—They are assertions or negations of dyadic predicates, written in infix notation. They consist of a certain number of words or variables, with a possible negation term in any position. They can also include a plausibility integer. Examples are:

```
JOHN IS MALE
JOHN IS FATHER OF JANE
*1 IS NOT FATHER OF *2
?80 WEATHER IS FINE
```

  The plausibility integers are numbers between zero and 100, zero corresponding to the negation of the assertion, 100 to its certainty, and 50 to its uncertainty. If the plausibility of an assertion is not given, it is assumed to be absolute. If the assertion is affirmatively worded, it is used in that form with a plausibility of 100. If the assertion is negatively worded, its negation is used with a plausibility of zero.

- Rules—Basically the rules are formal logic implications. A <- B is equivalent to A IF B, where A and B are assertive or negative clauses. Rules consist of two parts (premises and conclusion) joined by the IF symbol or its synonym.

A special case rule is the "axiom" or "fact," a rule without premises, that reduces to a single clause. Axioms may be considered as assertions or negations of dyadic predicates written in infix notation. Examples of axioms are:

- JOHN IS MALE—equivalent to the Prolog monadic predicate MALE(JOHN)
- JOHN IS FATHER OF JANE—equivalent to the Prolog dyadic predicate FATHER(JOHN, JANE)
- ?80 WEATHER IS FINE—indicates an 80 percent plausibility that the assertion is true
- * = *—an axiom that contains a variable and defines equality to AP998

Rules with premises allow the system to deduce new facts from the facts defined to it. Examples of rules with premises are:

```
*X IS SON OF *Y IF *X IS MALE AND *Y IS
   PARENT OF *X
*X IS PARENT OF *Y IF *X IS FATHER OF *Y OR
   *X IS MOTHER OF *Y
?70 I WILL GO TO THE THEATER IF WEATHER IS
   FINE
```

As the examples show, rules are accepted by the system in a way very similar to natural language. The last example can be read in the following way: "There is a 70 percent plausibility that I will go to the theater if the weather is fine."

When the conclusion of a rule depends on uncertain premises, the following are applied:

1. The plausibility of two premises separated by AND is the minimum of the plausibilities of the individual premises.
2. The plausibility of the conclusion of the rule is the product of the plausibility of the rule times the plausibility of the premises, divided by 100.
3. If the plausibility of the conclusion is smaller than a certain threshold value, and the subgoal answered by the conclusion included a variable, this solution is abandoned (i.e., its plausibility becomes zero).
4. If two premises separated by OR carry to the same conclusion, both results are passed to APL separately (as independent answers to the same question).

Structure of the knowledge base. AP998 maintains information in two different data spaces. The first one is a symbol table, where words are stored. The other is the rule table. The size of each is automatically chosen by AP998 to fit all of the words and rules defined to it. Their starting (minimum) size is 2K bytes. Their maximum size is 63K bytes.

A stack is also used for logic inferences, the size of which can be adjusted by the programmer within the same interval. (The default size is 2K bytes.) Therefore, the total data space for AP998 may vary between 6K bytes and about 190K bytes. The information in the stack allows AP998 to provide information on *why* it came to a given conclusion.

The maximum number of rules accepted by AP998 is about 3000. Of course, this number depends on the rules themselves, for rules are variable-length objects, depending on the number and sizes of their premises.

Example. As an example of the use of AP998, we will solve the following logic problem, taken from Reference 12:

"When Alice entered the forest of forgetfulness, she did not forget everything, only certain things. She often forgot her name, and the most likely thing for her

Figure 1 AP998 solution to logic problem

```
/* Solution to the ALICE problem in AP998 */
/* Definition of YESTERDAY                */
sunday    is yesterday of monday
monday    is yesterday of tuesday
tuesday   is yesterday of wednesday
wednesday is yesterday of thursday
thursday  is yesterday of friday
friday    is yesterday of saturday
saturday  is yesterday of sunday
/* Data about the lion and the unicorn    */
The lion    lies on monday
The lion    lies on tuesday
The lion    lies on wednesday
The unicorn lies on thursday
The unicorn lies on friday
The unicorn lies on saturday
/* Data about the phrases they said       */
The lion can say that on * if
    the lion lies on *
    and *Y is yesterday of *
    and the lion lies not on *Y
The lion can say that on * if
    the lion lies on *Y
    and *Y is yesterday of *
    and the lion lies not on *
The unicorn can say that on * if
    the unicorn lies on *
    and *Y is yesterday of *
    and the unicorn lies not on *Y
The unicorn can say that on * if
    the unicorn lies on *Y
    and *Y is yesterday of *
    and the unicorn lies not on *
/* Finally, both the lion and the unicorn */
/* have said that today, so that          */
Today is * if
    the lion can say that on *
    and the unicorn can say that on *
```

to forget was the day of the week. Now, the lion and the unicorn were frequent visitors to this forest. These two are strange creatures. The lion lies on Mondays, Tuesdays, and Wednesdays, and tells the truth on the other days of the week. The unicorn, on the other hand, lies on Thursdays, Fridays, and Saturdays, but tells the truth on the other days of the week.

"One day Alice met the lion and the unicorn resting under a tree. They made the following statements:

```
LION:    Yesterday was one of my lying days
UNICORN: Yesterday was one of my lying days
```

"From these statements, Alice, who was a bright girl, was able to deduce the day of the week. What was it?"

The solution is given by the AP998 program in Figure 1.

The APL2/PC product also includes a workspace containing a set of cover functions that can be used with the AP998 auxiliary processor. Figure 2 is a sample of their use in solving the Alice problem.

Performance. The performance of the auxiliary processor when compared against the use of pure APL functions depends on the application, on the number of rules, and on the average search depth to solve a question. In the case of the Alice example just detailed, the average time to solve the problem is 18.5 milliseconds on a Personal System/2* with a 25 Mhz processor speed. The APL2 function in Figure 3 needed 38 milliseconds to get the same result.

Of course, in this simple case, where the loop can be eliminated completely, the difference is not very large. In a real case, with many more rules and a true cascade of results, the use of the auxiliary processor would provide a real performance improvement.

Figure 2 Cover functions used to solve logic problem

```
      ASK 'TODAY IS *'
THURSDAY
      WHY
I HAVE USED RULE NUMBER 18:
  TODAY IS THURSDAY IF
      THE LION CAN SAY THAT ON THURSDAY
      AND THE UNICORN CAN SAY THAT ON THURSDAY
I HAVE USED RULE NUMBER 15:
  THE LION CAN SAY THAT ON THURSDAY IF
      THE LION LIES ON WEDNESDAY
      AND WEDNESDAY IS YESTERDAY OF THURSDAY
      AND NOT THE LION LIES ON THURSDAY
I HAVE USED RULE NUMBER 10:
  THE LION LIES ON WEDNESDAY
I HAVE USED RULE NUMBER 4:
  WEDNESDAY IS YESTERDAY OF THURSDAY
I HAVE USED RULE NUMBER 16:
  THE UNICORN CAN SAY THAT ON THURSDAY IF
      THE UNICORN LIES ON THURSDAY
      AND WEDNESDAY IS YESTERDAY OF THURSDAY
      AND NOT THE UNICORN LIES ON WEDNESDAY
I HAVE USED RULE NUMBER 11:
  THE UNICORN LIES ON THURSDAY
I HAVE USED RULE NUMBER 4:
  WEDNESDAY IS YESTERDAY OF THURSDAY
```

Figure 3 APL2 function

```
[0]  Z←ALICE;LL;UL;LC;UC;DAYS;YEST
[1]   DAYS←'SUND' 'MOND' 'TUES' 'WEDN' 'THUR' 'FRID' 'SATUR'
[2]   YEST←¯1ΦDAYS
[3]   (LL UL)←('MOND' 'TUES' 'WEDN')('THUR' 'FRID' 'SATUR')
[4]   LC←((~DAYS∈LL)∧(YEST∈LL)) ∨ ((DAYS∈LL)∧(~DAYS∈LL))
[5]   UC←((~DAYS∈UL)∧(YEST∈UL)) ∨ ((DAYS∈UL)∧(~DAYS∈UL))
[6]   Z←(LC∧UC)/DAYS
```

APL and neural networks

A neural network (also called a "connectionist system") is a set of elementary units, called neurons, mutually related by means of connections. Each neuron has a certain number of inputs and a single output, which can divide itself to provide connections (inputs) to many other neurons. In addition, a certain real number is associated with each neuron (its threshold) and with each connection (its weight).

The response of a neuron is a procedure that computes the output of the neuron as a function of its inputs, the weights of its input connections, and the threshold of the neuron. Usually, the response of a neuron can be expressed in the following way:

$$f((\Sigma w_i x_i) - \Theta) \qquad (1)$$

where x_i is the set of inputs to the neuron, w_i represents the respective weights of the input connections, Θ is the neuron threshold, and f is the response function.

If the response function f can only have the values zero or one, the neuron is called digital. Otherwise, it is called analogic.

In typical neural networks, all the neurons have the same response function, and connections are such that the neurons can be divided into a certain number of layers. Neurons in the first layer (the input layer) have inputs that do not come from other neurons, but that come from outside the neural network (from the environment). Neurons in the last layer (the output layer) have outputs that do not go to other neurons, but go instead to the environment. There may be zero to any number of intermediate layers (also called "hidden layers").

A neural network where at least one neuron sends a connection to another neuron in a preceding layer is a neural network with feedback. An interesting family of neuron networks with feedback is called "Hopfield neural networks."[13]

A neural network with just two layers (one input layer and one output layer) and no feedback between them is called a *perceptron*. In an important paper, Minsky and Papert proved that it is impossible to generate the "exclusive-OR" operation with a perceptron.[14] Their paper effectively put an end to all research in neural networks for several years. Current research usually uses neural networks with one intermediate layer.

Matrix representation of a neural network. In general, any neuron in a neural network can provide an input (a connection) to any other neuron. Therefore, the network structure can be represented by a square n-by-n matrix, where n is the number of neurons in the network and the element i,j in the matrix is the weight of the connection from neuron i to neuron j. Nonexistent connections can be represented as connections of zero weight (since Equation 1 is not affected by those null connections).

The connection matrix represents the structure of the network. To include all of the available information we need an additional vector with the thresholds of all of the neurons in the network, given, of course, in the same order as in the matrix rows and columns.

However, if the response of all of the neurons in a network is of the form indicated by Equation 1, the network will be equivalent to another network. In that other network, all of the neurons in the original network are present, with the same connections and

weights, but with zero threshold, and an additional input neuron, whose output is always one, has been added. The additional input neuron is connected to every neuron in the network by means of a connection whose weight is equal to minus the threshold of the target neuron in the original network. The proof of this assertion is obvious from Equation 1.

Thus, a neural network with n neurons and arbitrary thresholds can be considered equivalent to another neural network with $n + 1$ neurons, all of them with zero threshold. Therefore, the behavior of any neural network can be represented by a single matrix if the response of its neurons corresponds to Equation 1.

We will represent the inputs as a vector of values which we will extend to the same length as the number of neurons in the network. This extension is easy. It is enough to assume that all of the neurons have exactly one input, and assign zero as the input value of those neurons that in actual fact did not have any input.

The output of the network can be computed by means of the following simple APL2 function:

```
[0]  Z←CONEC COMPUTE1 INPUT;A
[1]   Z←INPUT
[2]  L:A←Z
[3]   Z←(INPUT+A+.×CONEC)>0
[4]   →(~A≡Z)/L
```

The left argument is the connectivity matrix that defines the network. The right argument is the input vector. Note that the response function, applied to the whole neural network, is digital, and reduces in this case to an inner product and a comparison.

The preceding function has a loop because each inner product propagates the effect of the input to the next accessible layer. The loop, which proceeds until the network stabilizes, will eliminate the transient stages and provide us with the steady-state result. In a neural network without feedback, the loop will be executed at most n times, where n is the number of layers in the network, usually equal to three.

Analogic neurons. The neurons described in the previous subsection were digital, since their output can only be zero or one. Analogic neurons can produce other outputs, such as any number in the [0, 1] interval. For example, a commonly used response function for neural networks is:

$$1/(1+e^{-\Sigma w_i x_i}) \qquad (2)$$

with appropriate corrections when the value obtained is too near one or zero. The following APL2 function computes the result of a neural network composed of neurons with this response function. The neural network is assumed to be represented by a single connectivity matrix.

```
[0]  Z←CONEC COMPUTE2 INPUT;A
[1]   Z←INPUT
[2]  L:A←Z
[3]   Z←÷1+⋆-INPUT+A+.×CONEC
[4]   Z[(Z<0.2)/⍳⍴Z]←0
[5]   Z[(Z>0.8)/⍳⍴Z]←1
[6]   →(~A≡Z)/L
```

Learning. We say that a neural network "learns" when it modifies its behavior in such a way that its response to a certain set of inputs adapts to another set of predefined "desired outputs."

Different learning procedures modify the weights of the connections of the neural network in such a way that the outputs get closer and closer to the desired values. These techniques require a teaching period during which the following steps happen:

1. One or several inputs are applied to the network.
2. The corresponding outputs are computed.
3. The outputs are compared to the desired outputs.
4. The weights of the connections are modified so that the outputs become more like the desired outputs.

The above process is repeated until the network behavior is acceptable.

One of the learning procedures most used in neural networks is called "back propagation" because the weight corrections are applied to those output neurons in the last layer that differ from the desired value, and then the correction is propagated to the preceding layers. The APL2 program in Figure 4 executes a version of back propagation.

This program makes use of several global variables: CONEC is the matrix defining the neural network. LAYERS is a vector that contains the number of

Figure 4 Back propagation program

```
[0]  BKPROP1 I;INPUT;OUTPUT;O;OUT;E;d;NT;NO;ER;N;E1
[1]   E←0.02
[2]   NT←1++/N←LAYERS
[3]  L:'Input value: ',▼IN[I;]
[4]   INPUT←1,IN[I;],(N[2]+N[3])ρ0
[5]   'Output value: ',▼OU[I;]
[6]   OUTPUT←OU[I;]
[7]  L1:O←(-N[3])↑OUT←CONEC COMPUTE INPUT
[8]   ER←0.5×+/(E1←O-OUTPUT)*2
[9]   →(ER<1E¯10)/0
[10]  d←E×OUT∘.×E1
[11]  NO←1+N[1]+N[2]+ιN[3]
[12]  d←d×CONEC[;NO]≠0
[13]  CONEC[;NO]←CONEC[;NO]-d
[14]  E1←(-NT)↑E1
[15]  NO←(∨/d≠0)/ι↑ρd
[16]  d←E×OUT∘.×(CONEC+.×E1)[NO]
[17]  d←d×CONEC[;NO]≠0
[18]  CONEC[;NO]←CONEC[;NO]-d
[19]  →L1
```

neurons in each layer. IN is a matrix of possible inputs. Finally, OU is the set of desired output values.

The program assumes that the number of layers in the network is three (the usual number). A few modifications would have to be done to apply a similar procedure to a perceptron or to a network with four or more layers.

Performance. In evaluating the performance of neural networks, there are two different considerations.

Performance of the learning process is one item. From the analysis of the back-propagation algorithm, it will be seen that the function contains an unavoidable loop. Therefore, the use of an interpreter (such as APL) will introduce a certain degradation. However, it must be remembered that the learning process is usually executed only once. After the neural network has learned successfully, it can be used many times without any further execution of the back-propagation algorithm or whatever else has been used. This means that the bottleneck is not so important unless the number of neurons is very large, and then APL may also have problems due to lack of space. But even this space problem can be solved, as the network connectivity matrices contain many zeros, and an implementation of sparse matrices can be used to make them fit in a given workspace.

Once the neural network has been trained, it will be applied to special cases, and this means that only the COMPUTE functions will be needed. It is easy to see that these functions also have a loop, but of a very different kind, since the number of times it is executed is equal to the number of layers in the network, which is usually equal to three. Therefore, interpretation time is negligible in this case as compared to the execution time of the inner product, where APL has no disadvantage as compared to a compilative program, since the inner product algorithm is a precompiled section of the interpreter.

APL and hypertext

The classical way of obtaining and presenting information is linear. In a book, or a written paper, or on the screen of a computer, the information is displayed as a succession of pages, each consisting of a number of lines, each line made of a succession of words. The reader will usually reach the desired information in a sequential process, by reading a word at a time, line by line, and page by page.

However, the use of certain "fast-reading" techniques allows the reader to browse the information in an extended way, overreaching the limits of the linear presentation. In an extreme case, rarely attained, we can consider that an ideally fast reader would be able to look at a page of text as a single unit, scanning it in a block and thus gaining a two-dimensional access to the information it contains.

What is hypertext? The term *hypertext*[15] has been applied to a recent means of information presentation that tries to transcend the limitations of the purely sequential display, allowing the reader a greater freedom in using scanning and retrieval procedures. The term was first applied in 1965 by Ted Nelson, who defined it as a hypothetical nonsequential writing tool.

All kinds of information can be combined to make up a hypertext application.

We can define hypertext as a nonlinear form of information presentation, where the units of information are the members of a hierarchy, linked in a certain way that makes it possible to attain very fast information retrieval. The search for an appropriate piece of data follows a nonlinear sequence directed by the train of thought of a reader, who is able to perform an associative navigation throughout the mass of information within reach. In this way, since it transcends the limitations of the written page, it can be said that hypertext provides the reader with a three-dimensional access to information.

The units of information in a hypertext system are usually the nodes of a hierarchical organization. The links that make up the hierarchy, which should be independent of the physical sequence of nodes, may be implicitly or explicitly defined by means of preprogrammed tags.

The benefits of hypertext are obvious. Besides the greater freedom provided to the reader by its three-dimensional access to information and its user friendliness, it is also quite easy to develop.

Hypertext media. All kinds of information can be combined to make up a hypertext application. We find:

- Visual information. This form is the most frequently used type in current computers. It consists of text, graphics, images, animation, video recordings, etc.
- Auditory information. This type includes speech and audio recordings.
- Other sensory data. At present, olfactory and tactile data are not usually found in computer applications, but perhaps in the future they will also be integrated into hypertext systems.
- Computer programs.

All of these kinds of information are kept in the ordinary physical storage media, such as fixed disks, diskettes, tapes, and compact discs.

Applications of hypertext. Hypertext methods can be applied wherever there is a need to manage large masses of information that can be divided into many chunks and accessed in a random way. For example:

- On-line documentation (help systems, reference works)
- Publishing (on-line dictionaries, computer-based encyclopedias)
- Computer-aided instruction (training manuals, tutorials, user guides)
- Expert systems, which require a highly developed interface to make use of the system so that it is friendly to a professional user who is not oriented to computer science (a physician, a lawyer, etc.)

Object-oriented programming and hypertext. Object-oriented programming (OOP)[16–18] is a programming method that is almost the exact opposite of classical procedural programming. In OOP, it is the *data* that are organized in a basic control hierarchy. One piece of data may be linked to another through a relation of descendancy, and this fact gives rise to a network (usually a tree) similar to the hierarchy of programs in procedural programming. There are also programs done in OOP, but they are appendages to the data (in the same way as in classical programming in which data are appendages of programs). It is possible to build global programs (accessible to all of the data in the hierarchy) and local programs (accessible from certain objects and their descendants).

In OOP, the execution of a program is fired by means of a *message* that somebody (the user, another program, or an object) sends to a given object. The recipient of the message decides which program should be executed. (It may be a local program or a global program which must be located through the network that defines the structure of the objects.)

Object-oriented programming is the appropriate way to program a hypertext application. In fact, the hierarchical data structure of OOP is the exact counterpart of the hierarchy of information units (the nodes) in hypertext. Hypertext links become the relations between objects in OOP. Hierarchical relations correspond to the links defining the hierarchy. Semantical relations provide the possibility of implementing other links that transcend the hierarchy.

The most generally used way to represent objects in object-oriented programming systems is by means of frames, a powerful data structure proposed by Minsky in 1975.[19] A frame system is a graph in which the nodes (frames) have a name and contain all of the information available about a given object. For example:

```
Frame TABLE
   Is_a: FURNITURE
   Files: 0,1,2
   Drawers: 0,1
   Legs: 4
   Light: 0,1
```

Object-oriented programming and APL2. In APL2, the existence of the general array makes it very easy to define and implement frames, which can be considered as general matrices of two columns, where the first element in each row contains a name and the second a (possibly multiple) value. For example, the frame mentioned above is a general matrix of five rows and two columns; it can be represented in APL2 in the following way:

```
TABLE ← 5 2 ⍴
   'IS_A' 'FURNITURE'
   'FILES' (0 1 2)
   'DRAWERS' (0 1)
   'LEGS' 4
   'LIGHT' (0 1)
```

With the use of frames, it is quite easy to build an object-oriented programming paradigm in APL2. Each object is represented as a frame, linked to other objects to form a hierarchy. The root of the hierarchy is called OBJECT and is initially defined as follows:

```
OBJECT ← 8 2 ⍴
   'PARENT' ''
   'CREATE'      'METHOD'
   'ERASE'       'METHOD'
   'PARENTS'     'METHOD'
   'CHILDREN'    'METHOD'
   'PROPERTIES'  'METHOD'
   'VALUE'       'METHOD'
   'METHODS'     'METHOD'
```

Each object in the hierarchy automatically inherits the properties and the methods defined by its ancestors (its parent and the ancestors of its parent), unless some property or method has been redefined, either by the same object or by a lower-level ancestor. The inheritance of methods and the ability to send messages to any object are easily implemented by means of the APL2 function MESSAGE, with the syntax:

```
MESSAGE 'Object' 'Method'
   [additional information]
```

and the implementation shown in Figure 5.

References 20 and 21 explain in more detail the applicability of APL2 for object-oriented programming. Thus, we can deduce that object-oriented programming in APL2 is a good way to program a hypertext application.

An on-line dictionary written in APL2 (OOP). A part of a Spanish on-line dictionary for the high-school level has been implemented in APL2/PC using object-oriented programming techniques. The dictionary currently contains the definitions of 2130 words in science and technology, distributed in the following fields:

- Biographies (123)
- Computer science (18)
- Technology (338)
  - Electronics (71)
  - Materials (59)
  - Vehicles (54)
  - Instruments (78)
  - Miscellaneous (76)
- Medicine (409)
- Biology (1100)
  - Anatomy (244)
  - Physiology (120)
  - Cytology and histology (38)

Figure 5 Implementation of MESSAGE

```
[0]   ΔR←MESSAGE ΔX;ΔOB;ΔMET;ΔSRCH;ΔA;ΔI;ΔB
[1]   ΔOB←' ' ELM↑ΔX
[2]   ΔMET←' ' ELM 2⊃ΔX
[3]   ΔX←2↓ΔX
[4]   →ΔE1 IF~EXIST ΔOB
[5]   ΔSRCH←ΔOB
[6]  ΔL:ΔA←(ΔB←GET ΔSRCH)[;1]
[7]   →ΔL1 IF(ρΔA)≥ΔI←ΔA⍳⊂ΔMET
[8]   →ΔE2 IF 0=ρΔSRCH←⊃ΔB[ΔA⍳⊂'PARENT';2]
[9]   →ΔL
[10] ΔL1:ΔX←(⊂ΔOB),ΔX
[11]  '' ⎕EA 'ΔR←',ΔSRCH,'_',ΔMET,' ΔX'
[12]  →0
[13] ΔE1:ΔMSG 'THE OBJECT' ΔOB 'DOES NOT EXIST. METHOD =' ΔMET
[14]  →0
[15] ΔE2:ΔMSG 'UNKNOWN METHOD' ΔMET 'FOR OBJECT' ΔOB
```

- Genetics (14)
- Biochemistry (78)
- Ecology (23)
- Paleontology (40)
- Microbiology (28)
- Zoology (incomplete) (302)
- Botany (incomplete) (140)
- Miscellaneous (73)

• Others (142)

The OOP application consists of a total of 2133 objects, three of which (the root of the hierarchy) are in the APL2/PC workspace, whereas the others (the words in the dictionary) are included in 44 files, created and used by means of the AP211 auxiliary processor.[22,23] The total size of these files is 1 372 216 bytes, which makes an average of 644 bytes per word definition, 31 187 bytes and 48 words per file. Words are distributed in the files thematically to reduce the overhead, since it can be assumed that groups of words searched in the dictionary will usually be related in this way. Therefore, not all files are equal in size, the largest one consisting of 142 words and 93K bytes, and the smallest one consisting of 8 words and 4K bytes.

Summary

This paper and others in the references show the usefulness of APL and APL2 for the most modern programming techniques and applications. Among these applications are artificial intelligence, neural networks, object-oriented programming, and hypertext, which have been described in some detail.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. J. A. Brown and M. Alfonseca, "Solution to Logic Problems in APL2," *SEAS Spring Meeting 1987*, Vol. 2 (1987), pp. 819–829.
2. *APL2 Programming: Language Reference*, SH20-9227, IBM Corporation (1987); available through IBM branch offices.
3. J. A. Brown, "APL2—A New Comer on the Artificial Intelligence Scene," *SEAS Spring Meeting 1986*, Vol. 2 (1986), pp. 819–830.
4. R. Guerreiro, "APL2 and LISP," *SEAS Anniversary Meeting 1988*, Vol. 2 (1988), pp. 1435–1460.
5. J. Ansell and M. Al-Doori, "Towards an Expert System," *APL-ication Proceedings* (1989), pp. 65–72.
6. D. Smellie and F. Evans, "A Structured Approach to Building Expert Systems," *APL-ication Proceedings* (1989), pp. 86–99.
7. A. Prys-Williams, "Expert Systems with Existing Software," *VECTOR* **5**, No. 3, 57–74 (January 1989).
8. P. Rodríguez, J. Rojas, and M. Alfonseca, "An Expert System in Chemical Synthesis Written in APL2/PC," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 4, 293–298 ACM, New York (1989).
9. K. Fordyce, J. Jantzen, G. A. Sullivan, Sr., and G. A. Sullivan, Jr., "Representing Knowledge with Functions and Boolean Arrays," *IBM Journal of Research and Development* **33**, No. 6, 627–646 (November 1989).
10. J. A. Brown, E. Eusebi, L. Groner, and J. Cook, *Algorithms and Artificial Intelligence in APL2*, Technical Report TR 03.281, IBM Corporation, Santa Teresa Laboratory, San Jose, CA (1986).

11. M. J. Tobar and M. Alfonseca, "Emulating Prolog in a PC APL Environment," *APL86 Conference Proceedings, APL Quote Quad* **16**, No. 3, 13–15, ACM, New York (1986).
12. R. Smullyan, *What Is the Name of This Book?*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978).
13. J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of Science* **79**, 2554–2558 (1982).
14. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA (1965).
15. *Hypertext: Theory into Practice*, Ray McAleese, Editor, Blackwell Scientific Publications, Oxford (1989).
16. B. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Co., Reading, MA (1986).
17. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
18. *Object-Oriented Computing*, G. Peterson, Editor, Vol. 1 & 2, IEEE Order No. 821 and 822 (1987).
19. M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, P. Winston, Editor, McGraw-Hill Book Co., Inc., New York (1975), pp. 211–217.
20. M. Alfonseca, "Object Oriented Programming in APL2," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 3, 6–11, ACM, New York (1989).
21. M. Alfonseca, "Frames, Semantic Networks, and Object-Oriented Programming in APL2," *IBM Journal of Research and Development* **33**, No. 5, 502–510 (September 1989).
22. *APL2 for the IBM Personal Computer*, Program Number 5799-PGG, PRPQ RJB411, Part No. 6242036, IBM Corporation; available through IBM branch offices.
23. M. Alfonseca and D. Selby, "APL2 and PS/2: The Language, the Systems, the Peripherals," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 4, 1–5, ACM, New York (1989).

Accepted for publication June 10, 1991.

Manuel Alfonseca *IBM Software Technology Laboratory, Paseo de la Castellana, 4, 28046 Madrid, Spain.* Dr. Alfonseca is a Senior Technical Staff Member in the IBM Software Technology Laboratory. He has worked in IBM since 1972, having been previously a member of the IBM Madrid Scientific Center. He has participated in a number of projects related to the development of APL interpreters, continuous simulation, artificial intelligence, and object-oriented programming. Eleven international IBM products have been announced as a result of his work. Dr. Alfonseca received electronics engineering and Ph.D. degrees from Madrid Polytechnical University in 1970 and 1971, and the Computer Science Licenciature in 1972. He is a professor in the Faculty of Computer Science in Madrid. He is the author of several books and was given the National Graduation Award in 1971 and two IBM Outstanding Technical Achievement Awards in 1983 and 1985. He has also been awarded as a writer of children's and juvenile literature.

Reprint Order No. G321-5453.