Frames, semantic networks, and object-oriented programming in APL2

by Manuel Alfonseca

This paper discusses the capabilities of APL2 for the implementation of frame systems and semantic networks, and for the use of object-oriented programming techniques. The fact that the frame is a basic data structure of APL2 makes this language very appropriate for the development of artificial intelligence applications using the indicated techniques. Examples are given of the way in which they may be applied to realistic situations.

Introduction

Since the beginning of computer science, computer programming has been an important and rapidly evolving discipline. The first programming systems were programs loaded by means of electrical connections physically plugged into movable switchboards. These were quickly replaced by other methods, easier to implement, that made it possible to attain higher levels of complexity. This evolution has changed not only the interfaces (paper type, punched card, teletype, teletypewriter, screen terminal, etc.) that introduce programs and data to the computer and the programming style (machine language, symbolic languages, high-level languages, application development systems, knowledge-based system environments, etc.), but also the logical *structures* that are used to organize information and data for manipulation by the program. In this paper, we discuss two of these new structures, *frames* and *semantic networks*, their use in a particular type of nonprocedural programming called *object-oriented programming*, and an implementation of these constructs in the APL2 programming language.

Procedural programming

For roughly the first thirty years, most programming was done with strictly *procedural* languages and techniques. Programs written in this way process information and data (the *data space*) by keeping it logically separated from instructions. The data space is organized according

to one or more *abstract data structures*, such as arrays, tables, queues, stacks, trees, or directed graphs.

These logical representations of data must be implemented internally by means of a smaller number of *basic data structures*, such as vectors or linked lists.

Certain high-level languages are well prepared to handle one or the other of these *internal types*—for instance, vectors in COBOL and FORTRAN, lists in LISP, and both in APL2. Moreover, it is always possible to build procedures to represent almost any abstract data structure on top of any internal structure. However, some internal types are better for certain abstract data structures. For example, vectors are specially useful to represent arrays and tables, while lists are used successfully to represent tables, stacks, queues, and trees.

New programming techniques

In the 1970s (disregarding a few languages for restricted applications which are even older), new programming procedures and techniques began to appear, particularly in relation to new research in the field of artificial intelligence. These techniques, which soon gave rise to new programming languages such as PROLOG, came to be known as *nonprocedural programming*. Finally, a certain subset of these techniques became known as *object-oriented programming*.

Together with the new techniques, new abstract data structures were introduced; these include logic structures (based on first-order predicate logic), scripts, conceptual dependency graphs, conceptual structures (after Sowa [1]), production rules, semantic networks, and frames. Here we describe in some detail the last of these two structures and the concepts of object-oriented programming.

- *Semantic networks*

Semantic networks were first defined by Ross Quillian in 1968 [2], as a consequence of his work on psychological models of human associative memory. A semantic network is a special case of a directed graph, frequently used in artificial intelligence systems as an appropriate way of representing knowledge.

A semantic network is a graph containing a certain number of *nodes* connected by a certain number of *branches*. The nodes represent *objects*, while the branches represent *relations* between objects. Both nodes and branches are labeled.

Several types of relation are possible. For example, there may be hierarchical relations, such as the following:

- *Is_a* indicates that one object is a subset of another object in the network, e.g., "Elephant is_a mammal".
- *Instance_of* indicates that one object is an element of another object in the network, e.g., "Jumbo instance_of elephant".
- *Prototype_of* indicates that one object is a special case or a model of another object.
- *A_part_of* indicates that one object is a physical part of another object, e.g., "Wheel a_part_of car".

There can also be relations that define properties, such as "Color," "Price," "Owner," or "Start_time."

One important concept in the operation of a semantic network is the *inheritance* of relations. Let us assume, for example, that relation R holds for objects A and B (in that order), and that relation S holds for objects B and C. Is there any relation between A and C? Of course, the answer to this question will depend on what is represented, respectively, by the relations R and S.

A particular case of relation inheritance is the *transitivity* of a relation, which can be defined thus: Let relation R hold for objects A and B; let relation R hold also for objects B and C. Does it then hold for A and C?

In a system based on semantic networks, decisions about the transitivity and inheritance of relations depend on the actual relations. Therefore, these systems usually provide a way to define the properties of relations in some particular programming language.

Semantic networks may be implemented using the linked list as the internal data structure. This explains the fact that LISP has been used as the programming language for many of these systems. However, as we show later, there are other data structures more appropriate than lists for representing a semantic network.

- *Frames*

Frames were proposed by M. Minsky in 1975 [3], in relation to his work on visual perception and natural language processing, as a data structure that tries to imitate the way in which human beings keep information in the brain and make use of it when the need arises. Like the semantic network, the frame is a special case of the directed graph.

A frame system is a graph in which the nodes (frames) contain all the information available about a given *object*, divided into a series of *attributes* or *slots*, each of which has a name (the name of a property of the object) and one or several values. A frame has also a name, by means of which all the information contained in the frame (all the slots) can be retrieved. For example,

```
Frame TABLE
    Is_a: FURNITURE
    Files: 0, 1, 2
    Drawers: 0, 1
    Legs: integer (default 4)
    Light: 0, 1
```

Once a given frame has been defined, it can be used to define new frames that will *inherit* automatically the

properties of the first frame. For instance,

Frame MY_DESK_TABLE
Is_a: TABLE
Files: 2
Drawer: 0
Light: 1

In the preceding example, certain properties are repeated (some slots have the same name as the parent frame TABLE), and they specify the properties of the new frame with a greater detail. Other properties may be missing (in this case the number of legs, which will be taken to be 4, the default value), or they may be new.

Frames use certain slots to generate a hierarchical structure similar to the one described for semantic networks, and properties may be inherited within this structure.

Semantic networks and frames differ in the following properties:

1. Properties of the data structure
   - In a semantic network, nodes hold no associated information except their names.
   - In a frame system, a node contains a great deal of information in the form of slots.
2. Properties of the relations
   - In a semantic network, all kinds of relations are automatically managed according to a set of methods of inheritance that can be defined by the programmer of the network.
   - In frame systems, there is usually a single class of relations that always allows the inheritance of properties.

However, the differences between semantic networks and frame systems are small enough that a system programmed for a semantic network may easily be used to build a frame system, and vice versa.

One of the most important properties of frame systems is the possibility of defining *actions* that should be performed under certain conditions—for example, to compute the actual value of a parameter or to produce secondary effects in the structure of the network whenever a slot value is requested and/or replaced. These actions are usually called *demons* or *attached procedures*, and are normally programmed as procedural programs in a high-level language.

- *Object-oriented programming*

How do classical (procedural) programming, nonprocedural programming, and object-oriented programming differ?

A procedural program (written in COBOL, FORTRAN, BASIC, PASCAL, LISP, or APL) comprises a certain number of sentences that execute sequentially in a predefined order that depends only on the values of the data being used by the program. This order can usually be deduced by visual inspection of the program.

A nonprocedural program (written in PROLOG, for instance) contains instructions that are not executed in any predefined order. They receive control from an *inference processor*, a procedural program that decides at every moment the order in which the sentences of the program should receive control (should be fired).

In both the procedural and the nonprocedural case, the basic unit of execution is the *program*. The data only provide values that will be used to perform computations or to decide the order of execution. A given application is a hierarchical set of programs (modules) each of which is capable of invoking other programs in the hierarchy. The data may be global (accessible from every program in the hierarchy) or local (accessible by the program to which they belong and, sometimes, by those at a lower level in the hierarchy).

In object-oriented programming, the environment is very different. Here it is the *data* that are organized in a basic control hierarchy. One datum may be linked to another datum through a relation of any type, and these relations give rise to a network (a tree or a graph) that resembles the hierarchy of programs in procedural programming. There are programs in object-oriented programming, but they are appendages of the data (as in classical programming data are appendages of programs). It is possible to build global programs (accessible to all data in the hierarchy) and local programs (accessible from certain objects and their descendants).

In object-oriented programming, the execution of a program is initiated (fired) by means of a *message* sent to a given object by the user, another program, or another 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 defining the structure of the objects).

The basic elements of object-oriented programming are *objects*, *messages*, and *methods*:

- *Objects* are the fundamental data structure, normally represented by means of lists or graphs and related to each other by means of abstract data structures such as frames or semantic networks.
- *Messages* allow objects to communicate with one another and with the external world.
- *Methods* are the procedures that should be executed when the messages are received.

The fundamental properties of object-oriented programming can be summarized thus:

- *Encapsulation*: All the information related to a given object is directly accessible from this object or its ancestors.
- *Inheritance*: Objects may inherit properties from other objects, and it is not necessary to repeatedly define a given property when it is shared by several objects.
- *Hiding*: Programs (methods) may be made local to certain objects and their descendants.
- *Message parsing*: Programming is performed by means of messages.
- *Dynamic binding*: Objects may be added and the hierarchy may be changed on the fly. Normally, this property requires the use of an interpreter rather than a compiler. This increases the flexibility of object-oriented programming, but lessens its performance.

Logical data structures in APL2

APL was noteworthy for the ease with which it could manage data with the logical structure of an array. However, its capacity for list processing was quite small, since it did not provide an efficient way to represent logical structures such as stacks, queues, trees, and directed graphs.

The APL2 language [4], designed by J. Brown, is a smooth extension to APL that incorporates a new basic data structure: the *general array*. Some of the ways in which this powerful data structure enhances the language and makes it appropriate for artificial intelligence applications and parallel programming have been described in [5] and [6]. However, the general array also makes it very easy to define and implement the most complicated data structures currently used in object-oriented programming applications in an extremely efficient way, far superior to any other programming language.

APL2 supports the following basic data structures:

- *Scalar*: A number or a character.
- *Simple array*: A rectangular collection of scalars in any number of dimensions. The simplest instance of a simple array is the vector, an array in one dimension.
- *General array*: A rectangular collection of simple arrays in any number of dimensions.

The general array is the most powerful data structure existing in computer languages today. It makes it very easy to represent very complex abstract data structures in a simple way. The following subsections present some examples.

- *Lists*

A list can be defined as a general vector of simple or general vectors. For instance, a list of three character strings would be represented in APL2 in the following way:

```
'String 1' 'String 2' 'String 3'
```

The above example is a general vector of three elements. All three elements are simple character vectors.

The APL2 expression

```
'String 1' ('String 2' 'String 3') 'String 4'
```

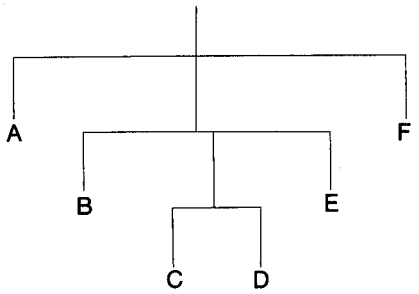
is also a general vector of three elements, and the first element, 'String 1', is a simple character vector; the second element, however, is another list, a general vector of two elements (the first of which is the simple character vector 'String 2', and the second, the simple character vector 'String 3'). The third element in the total structure is the simple character vector 'String 4'.

The above structure can also be considered as a list of three elements, the second of which is itself a list of two elements.

- *Trees*

A tree can be considered as a special case of a list, in which each element may be another list. Therefore, the preceding example is actually a tree.

The tree



is represented in APL2 in the following way:

```
A (B (C D) E) F,
```

where A, B, C, D, E, and F would be replaced by actual values in the terminal nodes of the tree.

- *Frames*

A frame can be considered as a general matrix of two columns, where the first element in each row contains the slot name and the second element the slot value. For example, the frame mentioned above,

```
Frame TABLE
   Is_a: FURNITURE
   Files: 0, 1, 2
   Drawers: 0, 1
   Legs: integer (default 4)
   Light: 0, 1
```

is a general matrix of five rows and two columns; it can be represented in APL2 in the following way:

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

where the ρ symbol is the APL2 function that creates an array, with the shape indicated by the left argument. In the example, TABLE becomes a matrix of five rows and two columns. Each element of this array is an array in itself (in this case, a vector or a scalar).

The example makes it clear that there is no difficulty in defining multi-valued slots. In fact, it can be said that APL2 is the only general-purpose high-level computer language that fully supports frames as a basic data structure.

The representation of demons, or associated procedures, is also straightforward. For instance, if a given frame needs to associate a pair of demons with a given slot (the first one receives control whenever the slot value is requested, the other one when the slot value is modified), the frame is defined as a four-column structure, where the third and fourth columns provide the names of the read and write demons, respectively, for each slot in the frame.

Semantic networks are not directly supported as APL2 basic data structures. However, since they are quite easy to represent by means of frames, it can be said that APL2 is also specially fitted for them.

Object-oriented programming in APL2

It is quite easy to build an object-oriented programming paradigm in APL2. The main data structure is the frame (which is a basic data structure in this language). The different objects in the frame system are linked to form a hierarchy. The root of the hierarchy is called OBJECT.

- *Attributes and methods*

Each object in the hierarchy has a number of different slots, belonging to the following classes:

- *Hierarchy definition slots*: These are called PARENT and CHILD.
  - PARENT always exists in every object. Its value is the name of the parent frame to this object, except in the case of the root of the hierarchy (frame OBJECT), where the value of the PARENT slot is empty.
  - CHILD exists unless this object is a leaf in the hierarchy (an object with no descendants). This is a multi-valued slot; i.e., its value is a vector of object names (the list of direct descendants of this object).
- *Method definition slots*: Their value is METHOD; their name is the method name. For each method M defined to an object O, an APL2 function called O_M describes the operation to be performed.
- *Property slots (attributes)*: Neither their names nor their values are restricted, except for those restrictions implicit in the preceding considerations.

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. (We define the level of an ancestor as the length of the path in the hierarchy between the ancestor and the object.)

The root frame (OBJECT) usually embodies the list of all methods that should be automatically applicable to all the objects in the hierarchy. The following is an example of the initial definition of the OBJECT frame:

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

It defines OBJECT as a frame with no parent, and provides the following methods for general use:

- **CREATE** creates a descendant of the object that receives the message to execute this method.
- **ERASE** erases the object that receives the message to execute this method.
- **PARENTS** generates the list of all the ancestors of the object that receives the message to execute this method.
- **CHILDREN** generates the list of all the immediate descendants of the object that receives the message to execute this method.
- **PROPERTIES** generates the list of all the properties known (including those inherited from ancestor frames), together with their corresponding values, for the object that receives the message to execute this method.
- **VALUE** obtains the value of a given property of the object that receives the message to execute this method. The property may be either owned or inherited.
- **METHODS** generates the list of all the methods (including those inherited from ancestor frames) available for the object that receives the message to execute this method.

- *Messages*

The ability to send messages to one object is easily implemented by means of the APL2 function

MESSAGE, which is described in the Appendix.

The syntax of MESSAGE is

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

where the presence or absence of any additional information, and the nature of this information, depend on the method being requested in the message. For example, in the methods ERASE, PARENTS, CHILDREN, PROPERTIES, and METHODS, described above, no additional information should be given. However, in the methods CREATE and VALUE, the following information is expected:

```
MESSAGE 'Object' 'CREATE' 'Child'
  [('Slot' 'Value') . . . ]
MESSAGE 'Object' 'VALUE' 'Slot'
```

In the case of CREATE, the optional information after the new child name represents a set of pairs Slot-Value that will be defined for the new object at the time of creation.

Examples of the use of MESSAGE are

```
      MESSAGE 'OBJECT' 'CREATE' 'JOHN'
        ('AGE' 50)
      MESSAGE 'JOHN' 'VALUE' 'AGE'
50
```

- *Other primitive operations*

Besides MESSAGE, five other primitive operations may be performed on objects, namely:

- **ASSIGN** 'Object' 'Slot' 'Value'

  If 'Object' does not contain an attribute named 'Slot', the attribute is created. In any case, 'Value' is the new value of attribute 'Slot'. In actual fact, 'Value' can be a list, as in the following example:

```
      ASSIGN 'JOHN' 'SONS' ('PHIL' 'TED'
        'NICK')
      MESSAGE 'JOHN' 'VALUE' 'SONS'
PHIL TED NICK
```

- **ERASE** 'Object' 'Slot' ['Value' . . .]

  If the list of values is not given, attribute 'Slot' is deleted from 'Object'. Otherwise, the corresponding values are deleted from the list of values of attribute 'Slot'. If, after deletion of the list of values, the value of 'Slot' is an empty list, the attribute 'Slot' is deleted from 'Object'. A list of values may be given only if the value of attribute 'Slot' is a list; for example,

```
      ERASE 'JOHN' 'SONS' 'TED'
      MESSAGE 'JOHN' 'VALUE' 'SONS'
PHIL NICK
```

- **INSERT** 'Object' 'Slot' 'Value' [. . .]

  The list of values given is added to the current list of values of attribute 'Slot'; for example,

```
      INSERT 'JOHN' 'SONS' 'PETER'
      MESSAGE 'JOHN' 'VALUE' 'SONS'
PHIL NICK PETER
```

- **TAKE** 'Object' 'Slot'

  The value, or the list of values, of attribute 'Slot' is passed back as the result of this function; for example,

```
      TAKE 'JOHN' 'SONS'
PHIL NICK PETER
```

  There is an important difference between TAKE and the VALUE method. The former only returns the value of attributes owned by the object, while the latter also provides the value if the attribute has been inherited from an ancestor.

- 'Object 1' **IS_A** 'Object 2'

  This function returns a 1 if 'Object 1' is a descendant of 'Object 2'. It returns a 0 in the opposite case.

A semantic network

It is easy to implement a semantic network in APL2 and use the methods of object-oriented programming to access, update, or erase the information contained in the network. The following program defines such a network, in this case one containing information about the subject matter in a curriculum or textbook.

```
[0]  CREATE_THEMES
[1]  INIT 'THEMES OOP'
[2]  ASSIGN 'OBJECT' 'WHO' 'METHOD'
[3]  MESSAGE 'OBJECT' 'CREATE' 'LIFE'
[4]  MESSAGE 'OBJECT' 'CREATE' 'WORLD'
[5]  MESSAGE 'OBJECT' 'CREATE' 'MAN'
[6]  MESSAGE 'LIFE' 'CREATE' 'BIOLOGY'
[7]  MESSAGE 'LIFE' 'CREATE'
       'MEDICINE'
[8]  MESSAGE 'BIOLOGY' 'CREATE'
       'BOTANY'
[9]  MESSAGE 'BIOLOGY' 'CREATE'
       'ZOOLOGY'
[10] MESSAGE 'BIOLOGY' 'CREATE'
       'MOLECULAR_B'
[11] MESSAGE 'BIOLOGY' 'CREATE'
       'ANATOMY'
[12] MESSAGE 'BIOLOGY' 'CREATE'
       'PHYSIOLOGY'
[13] MESSAGE 'BIOLOGY' 'CREATE'
       'HISTOLOGY'
```

```
[14] MESSAGE 'BIOLOGY' 'CREATE'
       'BIOCHEMIST'
       ('RELATION' 'CHEMISTRY')
[15] MESSAGE 'MEDICINE' 'CREATE'
       'PATHOLOGY'
[16] MESSAGE 'MEDICINE' 'CREATE'
       'MEDICAL_TEC'
       ('RELATION' 'TECHNOLOGY')
[17] MESSAGE 'MEDICINE' 'CREATE'
       'PRENATAL_M'
[18] MESSAGE 'MAN' 'CREATE' 'HISTORY'
[19] MESSAGE 'MAN' 'CREATE'
       'TECHNOLOGY'
[20] MESSAGE 'MAN' 'CREATE'
       'EDUCATION'
[21] MESSAGE 'WORLD' 'CREATE'
       'MATHEMATICS'
[22] MESSAGE 'WORLD' 'CREATE'
       'GEOLOGY'
[23] MESSAGE 'WORLD' 'CREATE'
       'ASTRONOMY'
[24] MESSAGE 'WORLD' 'CREATE'
       'PHYSICS'
[25] MESSAGE 'WORLD' 'CREATE'
       'CHEMISTRY'
[26] MESSAGE 'MAN' 'CREATE'
       'BIOGRAPHY' ('WORKS' 'METHOD')
[27] MESSAGE 'BIOGRAPHY' 'CREATE'
       'SCHWARTZCH'
       ('WORK' 'BLACK_HOLES')
[28] MESSAGE 'BIOGRAPHY' 'CREATE'
       'HOYLE'
       ('WORK' 'STEADY_STA')
[29] MESSAGE 'BIOGRAPHY' 'CREATE'
       'HAWKING'
       ('WORK' 'BLACK_HOLES')
[30] MESSAGE 'ASTRONOMY' 'CREATE'
       'SOLAR_SYST'
[31] MESSAGE 'ASTRONOMY' 'CREATE'
       'GALAXIES'
[32] MESSAGE 'ASTRONOMY' 'CREATE'
       'COSMOLOGY'
[33] MESSAGE 'GALAXIES' 'CREATE'
       'BLACK_HOLES'
[34] MESSAGE 'GALAXIES' 'CREATE'
       'QUASARS'
[35] MESSAGE 'COSMOLOGY' 'CREATE'
       'BIG_BANG'
[36] MESSAGE 'COSMOLOGY' 'CREATE'
       'STEADY_STA'
[37] MESSAGE 'BIOGRAPHY' 'CREATE'
       'NEWTON' ('WORK'
       ('OPTICS' 'GRAVITATION'))
[38] MESSAGE 'PHYSICS' 'CREATE'
       'OPTICS'
[39] MESSAGE 'PHYSICS' 'CREATE'
       'GRAVITATION'
[40] CLOSE
```

The preceding APL2 function was completely written using the object-oriented techniques and the functions MESSAGE and ASSIGN defined above. It creates a semantic network consisting of 38 nodes linked by means of the normal hierarchical relations PARENT/CHILD in the structure of a tree. At the same time, there are a couple of nonstandard relations (WORK and RELATION) linking certain nodes in the network. These relations are implemented as slots of certain objects. The slot name is the name of the relation, and the slot value is the list of names of the nodes in the network linked to this object by the relation. Multiple relationship is automatically supported by the fact that the value of an attribute may be a list.

The value of the WORK relation is the list of fields in which certain authors or scientists have worked. This relation has meaning only for the immediate children of the node BIOGRAPHY. Two methods have been built in this context, one of them to follow the relation in one direction, the second to follow it in the opposite direction:

- **WORKS.** This method generates the list of all fields in which an author can be said to have worked. To obtain the list, the method finds the value of the WORK attribute and adds to the resulting list the names of all the ancestors of the members of the list. Those nodes that would appear as ancestors of more than one member of the list are not repeated. The method is defined at node BIOGRAPHY, and in this way is available only for this object and its descendants.

  Example:

```
      MESSAGE 'HAWKING' 'WORKS'
BLACK_HOLES GALAXIES ASTRONOMY
  WORLD OBJECT
      MESSAGE 'NEWTON' 'WORKS'
OPTICS PHYSICS WORLD OBJECT
  GRAVITATION
```

- **WHO.** This method generates the list of all authors who have worked in a given field. To obtain the list, the method finds the names of all the authors who have worked in either the indicated field or one of its descendants.

  Example:

```
      MESSAGE 'ASTRONOMY' 'WHO'
SCHWARTZCH HOYLE HAWKING
```

```
      MESSAGE 'PHYSICS' 'WHO'
NEWTON
```

Both methods have been written in APL2 and are shown in the Appendix. (The RELATION relation is implemented in the same manner.)

Finally, the Appendix also includes an APL2 function that extracts information from the indicated semantic network using only the techniques of object-oriented programming.

Conclusions

The preceding considerations, as well as the simplicity of the programs listed in the Appendix, show that APL2 is a very appropriate language for object-oriented programming. The fact that frames are one of the basic data structures of the language makes this kind of application very natural and efficient.

The set of functions described above, together with a subsidiary set of auxiliary functions and four others embodying a windowing system, applicable both to frame systems and semantic networks, and supporting associated procedures (demons), have been implemented and are working successfully. The system (an APL2 workspace) runs both under APL2/PC and APL2/370 and supports the generation of frames and semantic networks, both as objects in the active APL2 workspace and as logical records in a file system specially created for this purpose.

The semantic-network example described in this paper has been expanded successfully to implement a hypertext application embodying an on-line dictionary. The total number of objects in the application is 2133, of which 2130 are contained in 44 files, while the remaining three objects (which make up the root of the hierarchy) are located in the main memory workspace. Each object in a file corresponds to the definition of a word in the dictionary. The total size of the files is about 1.3 megabytes, which corresponds to an average of 644 bytes per word. File management has also been implemented by means of the object-oriented paradigm. The application allows the user to display the information about each word in the dictionary, either through a direct call or by simply pointing to it anywhere. Mouse and window support is included in the hypertext application.

Appendix

- *Messages in APL2*

```
[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
```

- *Methods related to the WORK relation*

```
[0]    Z←BIOGRAPHY_WORKS OBJ;A;B
[1]    Z←''
[2]    →0 IF 0=ρA←TAKE(↑OBJ)'WORK'
[3]    →L IF 1≠≡A
[4]    Z←(⊂A),MESSAGE A 'PARENTS'
[5]    →0
[6]  L:B←MESSAGE(↑A)'PARENTS'
[7]    Z←Z,(↑A),(~B∈Z)/B
[8]    A←1↓A
[9]    →L IF 0≠ρA
```

The preceding APL2 function generates the result of the execution of the method in the variable Z.

- Line 2 gets the list of the values of the attribute WORK in the object receiving the message (OBJ). If this list is empty, the method is abandoned with an empty result.
- Otherwise, line 3 of the method tests to find whether WORK is a single-value attribute (when line 4 is executed) or a multi-value attribute (when control is passed to line 6 to execute a loop on the values).
- In both cases, lines 4 and 6 add to the result the list of all the ancestors of the values of the attribute WORK (the result of executing the method PARENTS against each of the objects in that list).
- Line 7 simply eliminates duplicates from the result.

```
[0]    Z←OBJECT_WHO OBJ;I;A;B
[1]    Z←''
[2]    A←TAKE 'BIOGRAPHY' 'CHILD'
[3]  L:→0 IF 0=ρA
[4]    →L2 IF 0=ρB←TAKE(↑A)'WORK'
[5]    →L1 IF(↑OBJ)≡B
[6]    →L1 IF OBJ∈B
[7]    →L0 IF 1=≡B
```

```
[8]   L3:→L2 IF 0=ρB
[9]    →L1 IF OBJ∈OBJECT_PARENTS B
[10]   B←1↓B
[11]   →L3
[12]  L0:→L2 IF~OBJ∈
      OBJECT_PARENTS⊂B
[13]  L1:Z←Z,A[1]
[14]  L2:A←1↓A
[15]   →L
```

The preceding APL2 function generates the result of the execution of the method in the variable Z.

- Line 2 gets the list of the direct descendants of the object BIOGRAPHY (i.e., the list of all the people who can work in any field). This list becomes the variable A.
- Lines 3 to 15 make a loop on the elements of A.
- Line 4 finds the list of works of one person in list A. This becomes list B.
- Lines 5 to 7 compare list B to the object receiving the message (OBJ, the field where we want to find the workers). If the object is in the list, control is passed to line 13, which inserts this person in the result variable (Z). Otherwise, a test is done to find whether OBJ is an ancestor of any element in B. If so, the corresponding element of A is also included in the result. Otherwise, it is ignored.

- *Object-oriented programming access to a semantic network*

```
[0]   THEMES
[1]   OPEN 'THEMES OOP'
[2]   'PARENTS OF HAWKING'
[3]   MESSAGE 'HAWKING' 'PARENTS'
[4]   'CHILDREN OF BIOGRAPHY'
[5]   MESSAGE 'BIOGRAPHY' 'CHILDREN'
[6]   'PROPERTIES OF HAWKING'
[7]   MESSAGE 'HAWKING' 'PROPERTIES'
[8]   'WORKS OF HAWKING'
[9]   MESSAGE 'HAWKING' 'WORKS'
[10]  'WHO WORKS IN BLACK_HOLES?'
[11]  MESSAGE 'BLACK_HOLES' 'WHO'
[12]  'WHO WORKS IN ASTRONOMY?'
[13]  MESSAGE 'ASTRONOMY' 'WHO'
[14]  'WHO WORKS IN OPTICS?'
[15]  MESSAGE 'OPTICS' 'WHO'
[16]  'WORKS OF NEWTON'
[17]  MESSAGE 'NEWTON' 'WORKS'
[18]  CLOSE
```

References

1. John F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publishing Co., Reading, MA, 1976.
2. M. R. Quillian, "Semantic Memory," *Semantic Information Processing*, M. Minsky, Ed., MIT Press, Cambridge, MA, 1968, pp. 227–270.
3. M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, P. Winston, Ed., McGraw-Hill Book Co., Inc., New York, 1975, pp. 211–217.
4. *APL2 Programming Language Reference*, Order No. SH20-9227, 1987; available through IBM branch offices.
5. James A. Brown and Manuel Alfonseca, "Solutions to Logic Problems in APL2," *APL Quote Quad* **17,** 356–361 (1987).
6. James A. Brown, Janice Cook, Leo H. Groner, and Ed Eusebi, "Logic Programming in APL2," *APL Quote Quad* **16,** 282–288 (1986).

Received May 24, 1988; accepted for publication August 1, 1989

Manuel Alfonseca *IBM Madrid Scientific Center, P. Castellana, 4, 28046 Madrid, Spain.* Dr. Alfonseca is Senior Technical Staff Member at the IBM Madrid Scientific Center. He joined the Scientific Center in 1972, and has since participated in a number of projects related to the development of APL interpreters, continuous simulation, artificial intelligence, and object-oriented programming. Results of this work are reflected in ten announced international IBM products. Dr. Alfonseca received the Electronics Engineering and Ph.D. degrees from Madrid Polytechnical University in 1970 and 1971, respectively, and the Computer Science Licenciature in 1972. He is a Professor of Theoretical Computer Science on the faculty of Computer Science in Madrid and is the author of several books. Dr. Alfonseca received the National Graduation Award in 1971 and two IBM Outstanding Technical Achievement Awards in 1983 and 1985. He has also been recognized as a writer of children's literature.