Advanced Data Structures in APL2/PC - Application to Object-Oriented Programming

By Manuel Alfonseca
IBM Madrid Scientific Center, P. Castellana, 4. 28046 Madrid (SPAIN)

Object oriented programming

What is the difference between classical programming, non-procedural programming and object oriented programming?

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

A non-procedural program (written in PROLOG, for instance) contains a certain number of instructions that will not be executed in a predefined order. They will receive control from an inference processor, a procedural program that decides in every moment the order in which the sentences of the program should receive control (should be fired).

In both the procedural and the non-procedural 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 where they belong and, sometimes, by those at a lower level in the hierarchy).

In object oriented programming, things are 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 this fact gives rise to a network (a tree or a graph) similar to the hierarchy of programs in procedural programming. There are also programs in object oriented programming, but they are appendages from the data (in the same way 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 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 defining the structure of the objects).

The basic elements of object oriented programming are the following:

- **Objects**, 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**, that allow objects to communicate with one another and with the external world.
- **Methods**, or 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. In this way, it is not necessary to define many times a given property if it is shared by a certain number of 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 makes it necessary to use an interpreter, rather than a compiler. This increments the flexibility of object oriented programming, but lessens its performance.

**Logical data structures in APL2**

APL was noteworthy by the ease with which it could manage data with the logical structure of an array. However, its abilities for list processing were quite small, so that there was not a proper and efficient way to represent logical structures such as stacks, queues, trees and directed graphs.

The APL2 language, designed by J. Brown, is a smooth extension to APL2 that incorporates a new basic data structure (the *general array*) that makes it very easy to define and implement the most complicated data structures currently used in artificial intelligence and object oriented programming applications in an extremely efficient way, far superior to any other programming language.

Thus, APL2 supports the following basic data structures:

- **Scalar**: A number of a character.
- **Simple array**: A rectangular collection of scalars in any number of dimensions. A particular case of simple arrays (the simplest instance) 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 complicated abstract data structures in a simple way. The following pages 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:

```apl
'String1' 'String2' 'String3'
```

The above example is a general vector of three elements. The first element is the simple character vector `'String1'`; the second is the simple character vector `'String2'`; and the third is the simple character vector `'String3'`.

The following APL2 expression

```apl
'String1' ('String2' 'String3') 'String4'
```

is a general vector of three elements. The first element is the simple character vector `'String1'`; the second is another list, a general vector of two elements (the first of which is the simple character vector `'String2'`; and the second is the simple character vector `'String3'`); the third element in the total structure is the simple character vector `'String4'`.

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

**Trees**

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

Let us look at the following tree:
It would be 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 the 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, which would be represented in APL2 in the following way:

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

The example makes it clear that there is no problem at all to define 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.

Semantic networks are not directly supported as APL2 basic data structures. But 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**

The APL2OOP system is a set of workspaces building an object oriented programming paradigm in APL2. The system runs both under APL2/PC and APL2/370, with slight differences affecting only the auxiliary processor interfaces. Two different versions of the system support the generation of objects, both in the active APL2 workspace, and as logical records in a file system specially applicable to this purpose.

The set of workspaces making up the full APL2OOP system is named as follows:

- **OOP**: Main workspace. Supports directly object oriented programming with the objects generated in the workspace.
- **OOP211**: Increment to OOP to support object oriented programming with the objects generated in a file system.
- **AP211H1**: Increment to OOP/OOP211 to emulate the AP211 auxiliary processor under APL2/370. It is required if the system will be executed under APL2/370 and the objects will be generated in a file system.
• **THEMEOOP.** Sample semantic network programmed using the APL2 object oriented techniques.

All the preceding workspaces are distributed in standard transfer form (name extension .ATF in APL2/PC, file type APLTF in APL2/370). To load them, the following system commands should be executed:

- Objects in the workspace:

  ```apl
  )CLEAR
  )IN OOP
  )IN THEMEOOP
  CREATE_THEMES
  ```

  The last two lines are necessary only if the sample semantic network will be loaded and generated.

- Objects in a file system:

  ```apl
  )CLEAR
  )IN OOP
  )IN OOP211
  )IN AP211H1
  )IN THEMEOOP
  CREATE_THEMES
  ```

  The last two lines are necessary only if the sample semantic network will be loaded and generated. The preceding line (copying AP211H1) is not needed in APL2/PC.

APL2/PC should be invoked with the AP211 and AP124 auxiliary processors. The former is not required if objects will be generated in the workspace.

**Data structures in APL2OOP**

The main data structure used by the system is the frame (which is a basic data structure in APL2). 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**: They 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` will exist 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` will describe the operation to be performed.

- **Property slots (attributes)**: there is no restriction to either their name or their value(s), except for those 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 that brings us from the ancestor to the object).

The root frame (`OBJECT`) will usually embody 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:
which 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 own 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.

**Associated Procedures (Demons)** Each slot in an object may include one or two associated procedures (sometimes called "demons") These are programs (methods) that will be invoked automatically under certain circumstances. The following types of associated procedures are supported by the APL2OOP system:

- Request associated procedure: it will be executed automatically whenever the value of the current slot is requested.
- Replacement associated procedure: it will be executed automatically whenever the value of the current slot is modified in any way (except for creation or deletion).

An object that has associated procedures in any of its slots will be represented by a 3 or 4 column frame. The third column of the frame contains the names of the request associated procedures, the fourth column contains the names of the replacement associated procedures.

The APL2 function executing one associated procedure must be programmed and defined before the corresponding associated procedure may take effect.

**Messages**

The ability to send messages to one object is easily implemented by means of APL2 function MESSAGE. 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, depends on the method being requested in the message. For example, in methods ERASE, PARENTS, CHILDREN, PROPERTIES and METHODS, described above, no additional information should be given. However, in 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:

```
MESSAGE 'OBJECT' 'CREATE' 'JOHN' ('AGE' 50)
```
or, if two demons are desired for the AGE slot:

```
MESSAGE 'OBJECT' 'CREATE' 'JOHN' ('AGE' 50 'DEMON1' 'DEMON2')
MESSAGE 'JOHN' 'VALUE' 'AGE'
50
```

**Other primitive operations**

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

- **ASSIGN** 'Object' 'Slot' 'Value'
  If 'Object' does not contain an attribute named 'Slot', it 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
```

To create a multivalued slot (a slot, the value of which is a list) with a single starting value (others may be added later by means of function INSERT) the following syntax should be used:

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

where the symbol used at the left of 'PHIL' is the APL2 "enclose" symbol (located in the keyboard in the Z key, in upper case).

When the value of a slot is being changed (i.e. when the slot is not being created) the replacement associated procedure (demon), if any, corresponding to that slot is automatically invoked. It is also possible to assign one or two associated procedures to a given slot by means of the ASSIGN function:

```
ASSIGN 'JOHN' 'SONS' ('PHIL' 'TED') 'RDEMON' 'WDEMON'
MESSAGE 'JOHN' 'VALUE' 'SONS'
(RDEMON EXECUTED HERE)
PHIL TED
```

- **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 the 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.

Example:

```
ERASE 'JOHN' 'SONS' 'TED'
MESSAGE 'JOHN' 'VALUE' 'SONS'
(RDEMON EXECUTED HERE)
PHIL
```

- **INSERT** 'Object' 'Slot' 'Value' [...]  
  The list of values given is added to the current list of values of attribute 'Slot'.  

When this function is executed, the replacement associated procedure (demon), if any, corresponding to that slot is automatically invoked.

Example:

```
INSERT 'JOHN' 'SONS' 'PETER'
(WDEMON EXECUTED HERE)
MESSAGE 'JOHN' 'VALUE' 'SONS'
(RDEMON EXECUTED HERE)
PHIL PETER
```

- **TAKE** 'Object' 'Slot'
The value, or the list of values of attribute 'Slot' is passed back as the result of this function.

When this function is executed, the request associated procedure (demon), if any, corresponding to that slot is automatically invoked.

Example:

\[
\text{T} \text{AKE } 'J\text{OH}N', 'S\text{ONS}'
\]

\[
(R\text{DEMON EXECUTED HERE})
\]

\[
P\text{HIL } P\text{ETER}
\]

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.

- 'Object1' IS_A 'Object2'
  This function returns a 1 if the left object is a descendant of the right object, and a 0 if this is not the case.

Object library management auxiliary functions

The following auxiliary functions are responsible of the lower level management of objects and object libraries. All of them work correctly regardless of whether the objects are located in the workspace or in a file.

- INIT 'Filespec'
  This function initiates an object oriented application. 'Filespec' is the name of the file to be used for the objects in this application. The name is ignored if the objects will be created in the workspace. This function creates the root of the hierarchical system of nodes (OBJECT).

- OPEN 'Filespec'
  This function is ignored if objects are in the workspace. It opens the file indicated if objects are in a file.

- GET 'Object'
  This function returns the full APL2 value of the indicated object, in the form of a frame.

- 'Object' PUT Frame
  This function creates the indicated object, assigning to it the indicated frame.

- EXIST 'Object'
  This function returns a 1 if the indicated object exists, a 0 if it does not.

- DELETE 'Object'
  This function deletes the indicated object.

- LIST
  This function lists the names of all the available objects in the current application.

- CLOSE
  This function is ignored if objects are created in the workspace. It closes the currently open library if objects are in a file.

The THEMEOOP sample 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, contained in the THEMEOOP workspace, defines such a network, containing information about the subject matters in a curriculum or textbook.

\[
[0] \text{CREATE THEMES}
[1] \text{INIT 'THEMES OOP'}
[2] \text{ASSIGN 'OBJECT' 'WHO' 'METHOD'}
\]
The preceding APL2 function has been completely written using the object oriented techniques and some of the functions (MESSAGE and ASSIGN) included in workspace OOP. 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 non-standard 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. 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 where certain authors or scientists have worked. This relation has sense only for the immediate children of node BIOGRAPHY. Two methods have been built in this context, one of them to follow the relation in one sense, the second one to follow it in the opposite sense.

- **WORKS**. This method generates the list of all fields where an author can be said to have worked. To obtain the list, the method will find the value of the WORK attribute and will add 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 'HAWKINS' 'WORKS'
BLACK_HOLES GALAXIES ASTRONOMY WORLD OBJECT
MESSAGE 'NEWTON' 'WORKS'
OPTICS PHYSICS WORLD OBJECT GRAVITATION

The following is the listing of the preceding method:

[0] Z←BIOGRAPHY_WORKS OBJ;A;B
[1] Z←'
[2] →0 IF 0=rA←TAKE(OBJ)'WORK'
[3] →L IF 1=rA
[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=rA

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

Example:

MESSAGE 'ASTRONOMY' 'WHO'
SCHWARTZ HOYLE HAWKINS
MESSAGE 'PHYSICS' 'WHO'
NEWTON

The following is the listing of the preceding method:

[0] Z←OBJECT_WHO OBJ;I;A;B
[1] Z←'
[2] A←TAKE 'BIOGRAPHY' 'CHILD'
[3] L:→0 IF 0=rA
[4] →L2 IF 0=rB←TAKE(+A)'WORK'
[5] →L1 IF(ÆOBJ)×B
[6] →L1 IF OBJ×OBJECT_PARENTS B
[7] →L0 IF 1=×B
[8] L3:→L2 IF 0=rB
[9] →L1 IF OBJ×OBJECT_PARENTS B
[10] B←1+B
[12] L0:→L2 IF¬OBJ×OBJECT_PARENTS×B
[14] L2:A←1+A
[15] →L

The next APL2 function extracts information from the indicated semantic network using only the techniques of object oriented programming.

[0] THEMES
[1] OPEN 'THEMES OOP'
[2] 'PARENTS OF HAWKINS'
[3] MESSAGE 'HAWKINS' 'PARENTS'
[4] 'CHILDREN OF BIOGRAPHY'
[5] MESSAGE 'BIOGRAPHY' 'CHILDREN'
[6] 'PROPERTIES OF HAWKINS'
[7] MESSAGE 'BIOGRAPHY' 'PROPERTIES'
[8] 'WORKS OF HAWKINS'
[9] MESSAGE 'HAWKINS' 'WORKS'
[10] 'WHO WORKS IN BLACK_HOLES?'
[11] MESSAGE 'BLACK_HOLES' 'WHO'
[12] 'WHO WORKS IN ASTRONOMY?'
[13] MESSAGE 'ASTRONOMY' 'WHO'
The window system

Workspace OOP also includes a set of four functions that make it possible to extract and update information in the frame system or semantic network in a user friendly way, by means of a window system.

**VIEW** is the entry point to the window system. When invoked, it shows a first screen with a window, located at the top-right corner, containing all the methods and special commands available for the current element of the network. The name of this element (OBJECT, by default) is shown at the bottom-left corner of the screen.

A method or a command is selected when the cursor is located in the corresponding line, and the ENTER key is pressed. If the method can be executed directly, its result will appear in a new window that replaces the one at the top-right corner. Press the Esc or Clear key to return to the main window. If the method requires extra information (as in the case of CREATE or VALUE) it will be requested from the user.

Command LIST displays the list of all the objects in the frame system or semantic network. The Esc or Clear key will return to the main window. However, if the ENTER key is pressed, the current object will be replaced by the one where the cursor is located. The main window will again receive control but, since the current object has changed, it is possible that the list of available methods will be different.

Command EDIT makes it possible to edit a method or any other APL2 function in the workspace. Remember that a method M defined to object O is named O_M. The Esc or Clear key exits the editor, and all changes done to the function will be lost. Other F-keys usable during edition are displayed at the bottom line.

At any time during execution of VIEW, except during edition of a method, the F1 key may be pressed. This defines a window located at the top-left section of the screen where APL2 lines can be executed. This window is made of two lines. Beware if the instruction being written does not fit in one line: the frame of the window should not be overwritten.

APL2OOP messages or primitive operations may be executed in the APL2 window. However, APL2 system commands cannot be executed in this window.

Once an APL2 line has been written, it will be executed by pressing the ENTER key. The result appears in the same window, expanded to the maximum size needed or available. After the result has been written, the ENTER or Esc key should be pressed before a new APL2 line can be input.

Pressing the Esc key at input time returns to the normal window state.

**Conclusion**

The preceding considerations, as well as the simplicity of the programs listed, 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, including a window system, is applicable to frame systems and semantic networks, has been implemented and is working successfully. The system (a set of APL2 workspaces) 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 applicable to this purpose.
References


