Data structures in the extensible programming language AEPL*

by E. MILGROM**

New York University
New York, New York

and

J. KATZENELSON***

Technion-Israel Institute of Technology
Haifa, Israel

INTRODUCTION

The extensible programming language AEPL has been designed as a tool for the implementation of a large class of problem-oriented languages or languages for specific applications. The reason for such a goal is that we believe that there exist numerous areas of human interest generating problems which can be solved with the aid of a computer. We think also that to be able to approach these problems using languages which are close to the terminology and the methodology of the respective areas is a significant advantage: it enables a user to think in familiar terms and it liberates him from the burden of extraneous detail. This has been the reason for the uneconomic proliferation of a large number of programming languages, each more or less well adapted to the solution of a particular class of problems (see for instance Sammet’s book14 for a survey of a number of problem-oriented languages). Extensible languages propose to cover wide areas of application at lesser cost and greater convenience. A detailed description of a large number of current extensible languages and systems can be found in a report by Solntseff,21 together with an extensive bibliography of the area. Many extensible language schemes have been described in detail.2-5,7,9,10,16,19

At the present time, we do not believe that the existing extensible languages can reasonably claim to replace all existing general-purpose and special purpose languages, mainly for reasons of efficiency. We concede therefore that the usefulness of AEPL will be greatest for application areas which do not warrant the cost of a specially written compiler and where the matter of efficiency is relatively unimportant. Another possible use of AEPL is during the design phase of a new application language: AEPL provides a rapid and cheap way to experiment with different versions of a proposed language.

We believe that the major innovations present in AEPL are the treatment of sets, used to create data structures and to define new data types, and the use of a powerful syntax description mechanism derived from the Markov Algorithm. We think also that most of the power of the system stems from its particular architecture and the concept of a special machine or processor which embodies the semantics of the language.

In what follows, we give a description of the data structure concepts of AEPL and we show how these concepts are used to create complex data structures and new types of data elements. A complete description of the language can be found elsewhere.15

The next section of this article presents some of the design objectives of AEPL; the following one describes in general terms the overall model of the AEPL system. Finally, the last section discusses the data structure concepts and the semantics of the data definition facility.

* Based in part on a thesis submitted by the first author to the Faculty of Electrical Engineering of the Technion in partial fulfillment of the requirements for the degree of Doctor of Science in Technology.

** Presently with the Department of Applied Mathematics, University of Louvain, 3030 Heverlee, Belgium.

*** On sabbatical leave at the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California 94720.
GENERAL DESIGN OBJECTIVES

During the design phase of AEPL, we tried to remain consistent with a number of general concepts and ideas which we discuss in this section.

Extensibility

The three main aspects of extensibility which we set out to provide were the ability to define new types of data items and new operations on old or new data items and the possibility to modify extensively the syntactic frame of the language. The AEPL system was designed so as to present itself to the users as a language, sometimes called core or kernel language, which includes a number of basic data types, a number of operators for these data types and a syntactic frame within which one can describe sequences of operations on data, i.e., programs. The core language includes also the tools which enable one to modify these basic constituents and create ‘extended languages’. Note, however, that the adjective ‘extended’ does not necessarily imply addition of features to the core language: one can use the extension mechanisms to produce a language which is less rich than the kernel by deletion of undesired features.

Minimality

In the design of an extensible language, one is tempted to limit the number of primitive language features to the bare minimum and to rely on extensibility for the creation of useful languages from the original core language. While the precise definition of a minimum set of features is a problem in itself, it is clear that the emphasis on minimality leads to kernel languages which are so primitive and involuted that their use is difficult: they have to be drastically extended in order to be of any practical use.

The design of AEPL is a compromise between a desire to keep the number of features of the kernel as low as possible and the requirement that the language be a fairly convenient programming tool.

Generality and completeness

Rather than to emphasize minimality, our approach has been to try to limit the number of primitive concepts, not the number of built-in language features. For that purpose, we tried to isolate a few very general ideas regarding data structures and syntax and to implement them in a language which would respect the concept of completeness as expressed by Reynolds: any value or class of values which is permitted in some context of the language should be permissible in any other meaningful context. This makes the language very regular: the number of special cases and particular conventions is greatly reduced. We believe that this is an important feature for an extensible language, since it reduces the number of possible inadvertent violations of the language rules.

THE MODEL

The AEPL system is composed of three parts:

- a core language,
- a processor,
- a translator.

1. The AEPL core language is a relatively small language which resembles Algol 60 in the sense that it includes a number of basic expression and statement forms (including declarations) and that the name-scoping of its variables is governed by an Algol-like block-structure. It differs from Algol 60 in the following aspects:

   - the primitive data items manipulated in AEPL are not the integer numbers, real numbers, arrays, etc., of Algol 60, but so-called t-values and objects as described below;
   - the AEPL core language contains a data definition facility which enables the user to define and manipulate new data structures;
   - the AEPL core language includes a number of facilities for modifying its own translator, thereby allowing an extensive syntactic variability.

2. The AEPL processor is a machine which operates on data structures of a particular kind, namely executable data structures called programs. Programs may be created and operated upon by the user in the same way as any other data structures. Programs are distinguished only by the fact that if the AEPL processor is applied upon them or, more precisely, if control is transferred to a program data structure, a number of actions will be performed by the processor.

The AEPL processor recognizes 63 different kinds of programs, i.e., the processor is a machine with a repertoire of 63 different instruc-
3. The AEPL translator is a program for the AEPL processor whose purpose is to transform an input string of characters into another data structure according to the rules of a special kind of grammar. At certain points of the translation, control may be transferred from the translator program to certain parts of the generated structure, thereby yielding "execution" of the transformed text by the processor.

The AEPL translator is composed of a lexical scan and a parsing phase. The parser consists of a parsing algorithm derived from the Markov Algorithm and a modifiable grammar which 'drives' the algorithm. The source text submitted by a user may contain statements whose execution affects the grammar by addition or deletion of rules. This feature is used to modify the syntax of the language: one may add new operators, new kinds of expressions, new types of statements dynamically; it is also possible to redefine (overload) or delete existing language structures.

4. In conclusion, one may view the AEPL system as consisting of a program (the translator) executed on a special machine (the processor). The translator transforms the input into several data structures. A certain number of those data structures can be interpreted as instructions for the processor and control can be transferred to them. If the input contains the appropriate command, the execution of the corresponding data structures by the processor will modify the translator: the language will have been extended.

The translator program is present in the memory of the processor together with the generated data structures unless those have been deleted by specific commands. Thus, at any instant of time, the "run time environment" of a user's program consists of the whole AEPL system augmented by the programs which were executed in the past and the data structures resulting from the execution of those programs. This approach is similar to that of languages such as LISP and BALM.

Since the processor is implemented conceptually as a program executed on an existing computer, the AEPL system can be considered to be interpretative.

DATA STRUCTURES

Principles

One of our aims has been to create in AEPL a simple but general data definition and manipulation facility which would allow us to handle a wide class of data structures. This facility should be powerful in order to enable the user to define complex data organizations; it should however be simple enough to understand and to use. This last point required that the data structure facility be based on a small number of well-chosen primitives.

Another design decision which has been made regarding AEPL is the total separation between data structures as conceptual organizations of data and storage structures or representations of data structures in memory. At present, the user is provided with a flexible data structure manipulation system, but he has no control over the way the structures are represented in memory.

It is clear that an algorithm can be specified and checked out for logical flaws without reference to memory representations. Indeed, when a complex algorithm is designed, it is common practice to clear the main issues and to avoid excessive detail by specifying the data structures first and postponing decisions regarding memory structures to a later stage. On the other hand, it is certain that the efficiency of any algorithm depends on the memory representations of the data structures. Therefore, in its current form, AEPL is a tool which is useful in the first stage of the design of algorithms. Using AEPL, one can verify and debug an algorithm in terms of its logic rather than in terms of its storage structures. After the debugging phase, however, it may be necessary to modify the default storage structures in order to increase the efficiency of the algorithm. At this stage, it is certainly easier to experiment with new storage structures, since one is at least almost certain that the logic of the algorithm is correct.

A complete programming system such as the one we aimed at should also provide means for controlling and checking the memory representations. This requires an implementation specification language which would allow the specification of storage structures by addition of statements to a program rather than by the modification of the program. This idea is not new: it has been proposed by Balzer, Schwartz, and Earley among others.

Basic data elements

There are two kinds of data elements in AEPL: terminal values (terminal values) and objects. Both kinds are strongly interrelated.
—t-values are entities which can be used as values (in the sense described below) of attributes of objects. Examples of t-values are integer numbers, character-strings, sets of integer numbers.

—Objects are entities to which six t-values are associated in the following way: we say that an object possesses six attributes, named respectively:
  name-, value-, mode-, type-, scope-, and rule-attribute.

Each attribute may possess a value, which is necessarily a t-value. If an attribute of an object possesses no value at some point in time, it is said that its value is undefined. It is possible to inquire about the value of any attribute of any object, and to modify that value.

Another way of looking at this would be to say that one object describes particular relationships between the six t-values which are the values of its attributes. The nature of these relationships will be explained below.

T-values

The AEPL system provides the following kinds of t-values:

—atomic t-values: integers, reals, character-strings, labels and references;
—compound t-values or, in our terminology, sets:
  —explicit sets or E-sets,
  —conceptual sets: C-sets, R-sets, P-sets, U-sets, I-sets, F-sets and primitive sets.

The primitive sets are:

—the set of all integer t-values,
—the set of all real t-values,
—the set of all character-string t-values,
—the set of all label t-values,
—the set of all reference t-values.

Although the term "set" is used, the concept is not in every case the same as the one used in mathematics. Some of the AEPL sets are ordered and may contain the same element many times; other sets (e.g., the primitive sets) correspond precisely to the mathematical notion of set: an unordered collection of distinct elements.

Classes of t-values

The set of all integer t-values is sometimes called the class of all integer t-values; similarly, the other primitive sets are primitive classes. The term "class" is used for a set which specifies the "kind" or "type" (in the Algol 60 sense) of a t-value. The primitive classes are available in the kernel language; other classes can be formed by means of the extension facilities. In fact, any set can be used in AEPL to define a class of t-values (see below).

Among the five primitive classes, only the class of reference t-values needs further explanation.

References

A reference is a t-value which designates an object in a unique way. One of the ways to gain access to the attributes of an object is by using a reference to that object. We do not concern ourselves with the implementation of such references: the important fact is that for every reference t-value there exists one and only one object which is referred to by that t-value. The reference concept is a generalization of the pointer concept which does not imply any particular implementation.

Sets

As mentioned above, a set, in AEPL, is a collection of t-values which is itself a t-value. Sets are used:

—to create aggregates of t-values,
—to define new classes of t-values.

AEPL distinguishes between two kinds of sets: explicit sets and conceptual sets.

An explicit set is a finite ordered collection of t-values which are effectively present in the system. Such sets correspond to the usual programming concepts of vector, list or sequence. An example is the explicit set composed of the integer t-values one, two and three, in that order.

A conceptual set is a collection of t-values which is defined implicitly. It may be finite or infinite, ordered or unordered. Such a set is defined by a predicate: it consists of all the t-values for which the predicate is true. In mathematical notation:

\[ \{ x \mid P(x) \} \]

An example is the set of all integer t-values, or the set of all character strings beginning with the letter A, or the set of all prime numbers smaller than 100. Sets are described in greater detail below.

Other classes of t-values

There are a number of classes of t-values which are not primitive classes, but which are used within the
translator program. Because of the model described above, the data structures of the translator are accessible to the user. Among other structures, the translator for the core AEPL uses a number of classes, called built-in classes, which define domains of t-values which may be of interest to the user: these classes are built in terms of the primitive classes in the same way that user-defined classes are constructed. Among these built-in classes is the class of all identifier t-values (character-strings beginning with a letter and containing only letters or digits) and the class of program t-values (explicit sets which can be interpreted as commands to the AEPL processor).

Objects and their attributes

Objects are entities to which six t-values are associated: one object describes specific relationships between these t-values, which are said to be the values of the attributes of that object. We describe here the roles of the attributes of an object.

The attributes of an object

The name-attribute

The value of the name-attribute of an object or, for short, the name of an object, is a t-value belonging to the class of identifiers: it may be used to refer to an object in the same way as a reference t-value. An identifier is thus associated with an object through the name-attribute of that object. Many objects may have the same identifiers as value of their name-attribute, but at every point in time a given identifier may be used to refer to only one of these objects. The choice of the object which is referred to by a given identifier is governed by the name scoping rules which depend on the block structure of the text submitted to the translator.

The value-attribute

The value of the value-attribute of an object or, again for short, the value of an object is a t-value whose class is defined by the mode-attribute of the object (see below). This attribute is closely related to the usual concept of value of a constant or of a variable in other programming languages.

The mode-attribute

The value of the mode-attribute of an object α is a reference to an object β whose value is a set of t-values to which the value of α belongs. The value of β thus defines the domain of the values of α or their class. For short, we say that β is the mode of α or that object α possesses mode β.

The reason for the existence of the mode-attribute is simply to allow the association of a meaning with the internal representation of the value of an object. The mode of an object α will indeed indicate whether the value of α is an integer t-value, a reference, a set, and so on. The corresponding Algol 60 concept is that of type of a variable; the name "mode" has been chosen because of the similarity with the Algol 68\textsuperscript{22} idea. Another purpose for the mode-attribute is its use, similar to that of syntactic type, during the parsing process. The modes of the objects involved in the parsing play indeed an important role in the selection of the grammar rules which must be applied to transform the input string into the parse tree.

The type-attribute

The type-attribute of an object can possess two values which indicate whether the object is a variable or a constant. An object is variable if the set of possible values for that object contains more than one element; otherwise it is constant. Clearly, one could indicate that an object α is constant by having its mode be a reference to an object β whose value is a set with one element, namely the value of α. However, it is usually preferable not to use this device; it is more appropriate to distinguish between a variable object whose value is the integer t-value seven and a constant object whose value is the integer seven by means of the type-attribute. The mode-attributes of these two objects could then both be a reference to an object whose value is the set of all integer t-values.

The scope-attribute

The scope-attribute of an object can possess three values denoted GLOBAL, LOCAL and DUMMY which define the scope of the relationship between the object and the identifier which is the value of its name-attribute.\textsuperscript{15,16}

The rule-attribute

The purpose of this attribute is related to the generation of program t-values by the parsing process.\textsuperscript{15,16} The value of the rule-attribute belongs to the built-in class of program t-values.
**Primitive objects**

To all primitive classes correspond built-in primitive objects. We thus have an object whose name is INT and whose value is the class of all integer t-values. The mode of this object has to indicate that its value is a primitive class: this is achieved by having the mode of object INT be a reference to a special object known to the system as the object whose name is PRIMITIVE and whose value is the set of all primitive classes. The value of the mode of PRIMITIVE is undefined. (The program which operates on the data structures recognizes the name PRIMITIVE.)

**An example**

Figure 1 illustrates, through a schematic representation, the relationships among three objects. The object A, i.e., the object whose name-attribute has the identifier A as value, has as other attributes:

- the value is the integer t-value twenty-seven (the dotted line is used to indicate this),
- the mode is a reference to the object INT,
- the type is the t-value indicating that A is a variable,
- the scope is the t-value indicating that the association of the identifier A with this object is global,
- the rule is irrelevant (its value is either undefined or not important in this context).

**Sets—detailed description**

**Explicit sets**

An explicit set or E-set is a finite ordered collection of t-values. It corresponds to the usual notions of vector, list or sequence. Every member of such a collection is called a component. E-sets may be used to create aggregates of data or to define new classes of t-values. The basic operations on E-sets are:

- selection of a component by ordinal position or by name (retrieval or storage of a value),
- addition or removal of a component,
- selection of a subset,
- test for membership,
- finding the number of components,
- concatenation of two E-sets.

The language possesses a notation for constant E-sets, e.g.,

\[ E\{1,2,'ABC','E\{3,4}\} \]

which denotes an E-set of four components, the fourth of which is itself an E-set with two components.

One can define other operations on E-sets in terms of these basic operations by means of the syntactic extensibility mechanism of the language.

If one wishes, for instance, to introduce unordered finite sets of distinct elements, one can do so easily by representing these sets as E-sets and by ignoring the ordering relation among the components. At least two basic operations must however be redefined:

- the test for equality between two sets must ignore the ordering,
- the addition of an element to a set must verify that the element is not yet a member of that set.

Other operations on unordered sets (union, intersection, power set, and so on) can then be written in terms of the basic operations. Unordered sets of ordered pairs may be used, as in SETL, to represent mappings; functional application can then be easily defined for such mappings.

E-sets can be used to define new classes of t-values by enumeration. For instance, the set E\{‘1’,‘3’,‘5’, ‘7’,‘9’\} could be used to define the class of odd digits. Similarly, the set E\{1,2,3,5,7,11,13\} could be used to define the class of prime integers smaller than 15.

**Conceptual sets**

A conceptual set is a set defined by a predicate. Such a set is not present in the system under the form of a collection of t-values: it is present purely by convention as the set (in the mathematical sense) of t-values for which the predicate is true. A conceptual set is thus a collection of t-values defined by a certain common property. These sets are represented in AEPL by *descriptions*.
of the properties of their elements rather than by a list of their elements. Since such a description is usually composed of several elements, AEPL represents a description by an E-set. The description of a conceptual set may be stored in the value-attribute of an object; the mode of that object will indicate that its value may be interpreted as the description of a conceptual set.

The primitive sets are conceptual sets corresponding to the primitive classes of AEPL: they exist in the system as the values of the primitive objects INT, REAL, CHAR, LABEL and REFERENCE. Other conceptual sets belong to one of the following categories: C-sets, R-sets, P-sets, U-sets, I-sets and F-sets.

The reason why there is more than one kind of conceptual set besides the primitive sets is simply one of ease of programming: it is not always convenient to represent a set by a general predicate; certain particular cases deserve special treatment.

The basic operation involving conceptual sets is the test for membership.

C-sets

According to the functions of the attributes described above, if an object \( \alpha \) has an E-set as value (i.e., as value of its value-attribute), then the mode of \( \alpha \) should be a reference to an object \( \beta \) whose value is a set of E-sets, namely the class to which the value of \( \alpha \) belongs. This class may be defined by a C-set: a C-set is indeed a set of E-sets. Its description is composed of the following five t-values:

- **Number-type** is a t-value which indicates whether the number of components of the E-sets which belong to this C-set is variable or constant.
- **Number** is a t-value which is the number of components of the E-sets which belong to this C-set if this number is constant (examine number-type to find this out); otherwise, this t-value is a reference to a Boolean function of two arguments: an integer t-value \( n \) and a reference to an object \( \beta \) whose value belongs to the class of E-sets described by this C-set. The function returns the value true if and only if the integer \( n \) is a permitted value for the number of components of the value of \( \alpha \) belongs.
- **Component-type** is a t-value which indicates whether the E-sets belonging to this C-set are homogeneous or not. A homogeneous E-set is one whose components belong to the same class.
- **Component-class** is a t-value which defines the class of every component of any E-set belonging to this C-set in the following way. If the E-sets are homogeneous (examine component-type to find this out), then component-class is a reference to an object whose value is the class to which all the components belong. If the E-sets are nonhomogeneous, then this t-value is a reference to a function of two arguments: an integer t-value \( n \) and a reference to an object \( \alpha \) whose value belongs to the class of E-sets described by this C-set. The result of this function is a reference to an object whose value is the class to which the \( n \)th component of the value of \( \alpha \) belongs.
- **Names** is either undefined or a reference to a function of two arguments, an identifier \( id \) and a reference to an object \( \beta \) whose value belongs to the class of E-sets described by this C-set. The function returns an integer t-value \( n \) which is the ordinal position of the component of the value of \( \alpha \) whose name is to be \( id \). If no such component is found, then the function returns zero.

Figure 2 schematizes an example in which the object PAIR has, as its value, the set of all E-sets with two integer components which are unnamed. The values of number-type (constant), number (2), component-type (constant), component-class (a reference to INT) and of names (undefined) define this by convention. The value of object X (a pair of integers) belongs to the class defined by PAIR, so the mode of X is a refer-
In this article, we have presented an overview of the main features of the AEPL system. We have discussed in detail the data structure concepts which form the basis for the data definition facility of AEPL. Using these concepts, a user can create, in a straightforward manner, new kinds of data items and aggregates of data. Other features of the system enable the user to define new operators for any kind of data items and to create new language structures such as statement forms.

Our experience with the language has been limited to pen-and-paper coding since the system is not yet implemented. A language for the creation and manipulation of linear graphs obtained by extension of the
kernel AEPL is described in detail elsewhere. In our opinion, this and other examples show the feasibility and the usefulness of the approach described in this paper.

ACKNOWLEDGMENTS

We would like to express our gratitude to the following people for the numerous and fruitful discussions we had with them: Prof. Y. Wallach and Dr. E. Kantorowitz of the Technion, Prof. J. Feldman of Stanford University, Profs. J. C. Boussard and M. Griffiths of the University of Grenoble and Messrs. S. Schuman and P. Jorrand of the IBM Scientific Center in Grenoble.

REFERENCES

1 R M B ALZER
Dataless programming
Proc AFIPS 1967 FJCC pp 535-544
2 J R BELL
The design of a minimal expandable computer language
Doctoral dissertation Stanford University 1968
3 T E CHEATHAM Jr
The introduction of definitional facilities into higher level languages
Proc AFIPS 1966 FJCC pp 623-637
4 T E CHEATHAM Jr A FISHER P JORRAND
On the basis for ELF—An Extensible Language Facility
Proc AFIPS 1968 FJCC pp 937-948
5 C CHRISTENSEN C J SHAW (eds)
6 J EARLEY
Toward an understanding of data structures
Comm ACM 14 10 (Oct 1971) pp 617-627
7 B A GALLER A J PERLIS
A proposal for definitions in Algl
Comm ACM 10 4 (April 1967) pp 204-219
8 B A GALLER A J PERLIS
A view of programming languages
Addison-Wesley Publ Co Reading Mass 1970
9 M C HARRISON
BALM—An extendable list-processing language
Proc AFIPS 1970 SJCC pp 507-511
10 E T IRONS
Experience with an extendable language
Comm ACM 13 1 (Jan 1970) pp 31-40
11 J KATZENELSON
The Modified Markov Algorithm as a language parser—linear bounds
J of Systems and Computer Sciences to be published
12 J KATZENELSON E MILGROM
The Markov Algorithm as a language parser
In preparation
13 A A MARKOV
Theory of algorithms
Academy of Sciences of the USSR 1954 English translation by Israel Program for Scientific translations
14 M D MelROY
Macroinstruction extension of compiler languages
Comm ACM 3 4 (April 1960) pp 214-220
15 E MILGROM
Design of an extensible programming language
Doctoral dissertation Technion—Israel Institute of Technology 1971
16 E MILGROM J KATZENELSON
AEPL—An Extensible Programming Language
In preparation
17 J C REYNOLDS
GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept
Comm ACM 13 5 (May 1970) pp 308-318
18 J E SAMMET
Programming languages: history and fundamentals
Prentice-Hall Co Englewood Cliffs N J 1969
19 S A SCHUMAN (ed)
20 J T SCHWARTZ
Abstract algorithms and a set-theoretic language for their expression
Preliminary draft Courant Institute of Mathematical Sciences New York University New York N Y 1970-72
21 N SOLNTSEFF A YEZERSKY
A survey of extensible programming languages
Computer Science Tech Report No 71/7
MacMaster University Hamilton Ontario 1971
22 A VAN WIJNGAARDEN (ed)
Report on the Algorithmic Language ALGOL 68