An Introduction to the ADAMS Interface Language
Part I

John L. Pfaltz
James C. French
Andrew Grimshaw

IPC-TR-91-06
April 17, 1991

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 2290l

This research was supported in part by
DOE Grant #DE-FG05-88ER25063 and
JPL Contract #957721

Abstract

ADAMS provides an interface between host application programs and a space of persistent data items, or a database. The interface consists of ways to describe a user's space of persistent data items, sometimes called a "data description language", or DDL, together with mechanisms to access those items, which is sometimes called a "data manipulation language", or DML, or a "query language". In this part we concentrate only on the descriptive power of ADAMS. What kinds of databases can be represented in ADAMS. In Part II, we will focus on the manipulative and query power of ADAMS.

Every database implementation is eventually grounded in a number of primitive concepts which are fundamental to understanding the system. ADAMS is based on six primitive concepts: attribute, codomain, element, map, sequence, and set, which we regard as fundamental to all database implementation. We show that ADAMS has at least the same descriptive power as the more traditional database models, such as the relational model, the semantic model, and the object-oriented model by showing all such database configurations can be expressed in terms of these primitives.

ADAMS, the Advanced DAta Management System of the University of Virginia's Institute for Parallel Computation, has been designed to function as clean interface between user processes and one, or more, persistent data bases. It is not meant to serve as a traditional database language. In particular, it provides no mechanisms for displaying retrieved data. We expect that user programs will be written in some established language (and henceforth called the *host language*), such as Fortran, C, or Pascal, that is most appropriate for the desired data processing operations. ADAMS, itself, only provides mechanisms for defining and naming persistent database structures on permanent storage media, and for accessing these structures. Thus, ADAMS can be regarded as a tool for implementing database access mechanisms (possibly other database languages) and for porting them to a variety of hardware configurations, including parallel machines.

An interface is really a language — a way of referencing those things of importance to the interface. In this paper we concentrate entirely on the linguistic aspects of the interface and largely ignore the mechanisms we have used to implement it. All languages are designed to talk about some area of interest; one cannot understand the structure of the language without first understanding what it is about the world that the language was designed to convey. ADAMS, for example, has a structure that is quite different from most familiar programming languages because ADAMS was not designed to describe computational processes. The ADAMS language is not concerned with data processing. Its entire focus is on the storage and retrieval of "things" (essentially character, or bit, strings).

In the first sections we describe, using a variety of intuitive analogies, the primitive concepts that we think are fundamental to a database interface. Then we describe the actual structure of the ADAMS language — the syntactic constructs it has adopted to describe and manipulate these basic concepts. Finally, we show how higher level database concepts, as developed in the relational, semantic, and object-oriented models, can be expressed in this interface language. In effect, this last section illustrates the expressive power of the ADAMS language and how it can be used in the world of computer applications that are highly data dependent.

1. Primitive Concepts in the ADAMS Approach

Basically, ADAMS provides a systematic way of dealing with *sets* associated with a database. But to do so, it must identify three very different kinds of sets and provide three different linguistic mechanisms to deal with each kind.

First, there are abstract sets of *values*, where a value denotes some bit string that will be interpreted according to a pre-defined convention. For example, the set of "integers", the set of "reals", or the set of "strings of 5 or fewer characters". Many programming languages call these sets "data types". We will call them *codomains*, and discuss them more thoroughly in later sections.

The second, and most important, ADAMS set concept is that of a *set of elements*, where an element is considered to be any of a large variety of instantiated objects. Typically, elements are fairly simple composite structures with several associated values; but they can be arbitrarily complex structures. ADAMS provides computational procedures to manipulate sets, or elements; and

the term *set* will only be applied to sets of instantiated elements. In the ADAMS language the word "set" will always refer to one of these.

The elements of a set are unordered. a closely related concept is that of a *sequence* of elements on which a user specified order has been imposed. Given any element of a sequence, there is a unique successor and a unique predecessor element.[1] Sets and sequences, in ADAMS, are treated very similarly, except for the kinds of operations that can be defined on them. The union of two sets is meaningful. The union of two sequences is not.

Finally, there are abstract sets of ADAMS elements. An ADAMS *class* is a generic pattern that defines the characteristics and properties of *all possible* elements of that kind. Thus it denotes the universe, U_{class}, of all such elements.[2] Each ADAMS element is only a single representative of its class. A database, for example, may contain instances of "linked lists", or instances of "relations" with different schema. ADAMS provides a language in which one can describe such different classes of elements.

Individual *codomains* and *classes* are defined "syntactically" within the ADAMS language. Specific *sets* and *sequences* (of elements) must be defined by construction (e.g., by computational process).

The underlying premise of ADAMS is that there are only a few very basic underlying concepts in database representation. We assume that all database models can be expressed in terms of these basic concepts, and thus all databases can be implemented using this reduced set of database concepts. Moreover, we assume that if one knows how these few concepts have been represented, then one knows how the entire database has been implemented.

The primitive concepts are

a) codomain - abstract set of data types
b) class - abstract set of element typles
c) element - instantiated ADAMS object
d) set - " " "
e) sequence - " " "
f) attribute - functional operator/instantiated ADAMS object
g) map - " " " " "

where each of these will be described in detail in the next sections. In the last sections we illustrate how more traditional database models can be expressed in terms of these ADAMS concepts.

1.1. Elements

An *element* is a primitive concept in ADAMS. An element has no *a priori* form or structure. The only essential characteristic of an element is that it is *uniquely identifiable.* We will give two analogies to expand on the "element" concept.

---

[1] ADAMS is a pure implementation of the Peano "successor" concept, except that, since we are concerned with the persistent *storage* of such sequences, all sequences must be finite. Consequently, all sequences have a unique first and last element (which respectively have no predecessor or successor). In UNIX, a sequence of objects is called a *stream*, which is processed by filters in pipes. An ADAMS sequence can be regarded as a database implemented stream.

[2] In many object-oriented systems [GoR83, Kim90] the class denotes not only the universe of possible elements in the class, but the universe of actually instantiated objects as well. This is an effective mechanism when using the object-oriented paradigm as a programming language; it causes considerable difficulty when using the object-oriented paradigm for database retrieval.

In formal set theory, one traditionally defines a set as a "collection" of elements (this primitive definition is always a bit circular). It is irrelevant what these elements "really are"; all that is required is that we be able to identify them, say as *a, b,* or c, so that we may make statements such as S = { *a, b* } or $c \in X \cap Y$. These elements are certainly not the strings "a", "b", or "c" themselves. And it makes no difference whether we ever know what the symbols *a, b* or *c* denote. "Meaning" may, or may not, later be ascribed to them.

A different, more concrete, analogy comes from the area of object-oriented programming. An "element" may be regarded as an *object*. An instantiated object is identifiable; it exists in a virtual address space and its *address* uniquely identifies it. Being able to identify an object, that is knowing its "address", tells us nothing about the object itself. It may be a chunk of executable code, a linked list, a single data value, or a text file.

The concepts of *element* and *object* are so nearly synonymous that we are tempted use them interchangeably. We have chosen to use the term "element" partially because the term "object" already has many existing connotations in the computer literature [BBK87, CoM84, SSE87], and partially because ADAMS elements are not properly objects in the sense of [Str88]. Nor is ADAMS is an object-oriented database system in the sense of [Kim90]. For example, there are no methods associated with ADAMS elements. "Object-based", as in [Weg87], is probably the best description of ADAMS, for it does employ the usual class hierarchy, with multiple inheritance [Car84]; and it can support object-oriented database languages.[3]

All elements (or instantiated objects, if you prefer this terminology) in an ADAMS database must be *typed*, that is belong to a known *class.* Except for the four system defined classes[4],

SET,
SEQUENCE,
ATTRIBUTE, and
MAP,

the class (or type) of an element has no pre-defined significance. A user can define arbitrary classes, such as "Q", "Budweiser", or "XX3", and instantiate elements belonging to these classes. Any significance attached to a user defined class is completely the responsibility of the user. In section 2.1 we will discuss the way that classes are defined. ADAMS only requires that:

(1) every element in the system must be assigned some class (or type) at the time of its instantiation, and
(2) if the element is to function as either a set, sequence, map, or attribute (as described in the next sections), it must be so classified.

1.2. Names and Unique Identifiers

All ADAMS elements are uniquely identifiable. The unique identifier of an element *x* we denote by uid_x, or just its *uid*. Every element has a *uid*, but such identifiers are commonly various forms of "virtual addresses" that are not accessible outside of the particular database structure in which the element is embedded; *uid*s are "invisible" to user programs. But, unless some ADAMS elements are "externally identified", or "named", subsequent access will be impossible. As an example, consider a file of records. The individual records in the file need not be named, but the file itself must be named so that it can be attached to processes using it.

---

[3] The ADAMS run-time system has also been implemented in C++, but this is irrelevant to the question.

[4] These four pre-defined classes, together with CLASS itself, are super-classes of which all user defined classes are sub-classes.

The way that ADAMS uses *names*, or literal identifiers, is at the syntactic heart of ADAMS. All ADAMS names are recorded in a dictionary, in which existing names and their meaning[5] can be found; and into which newly defined names are entered. Because, ADAMS is a "permanent" entity, its philosophy is quite different from more traditional programming languages. Declarations made several years ago in one process may be saved in its persistent dictionary, and used repeatedly by other processes. Consequently, the names used to designate classes, codomains, and elements are themselves persistent and may not be freely reused in separate programs.

While elements are nameable; most elements are never, in fact, given unique literal names. One reason that most instantiated elements remain unnamed is that, for all intents and purposes, only a relatively small name space is available. Good programming (and database design) practice dictates that names be mnemonic and relatively short. One, time honored way that mathematicians extend a restricted name space is through subscript notation.

The name, or literal identifier, of any *instantiated* element may be *subscripted*. Thus the names of elements, sets, sequences, attributes, and maps may be subscripted; but the names of generic sets (that is, classes and codomains) may not. Subscripting an identifier does not imply an underlying array, or vector, structure, as in most languages. Only the identifier is subscripted. However, regular array and/or vector structures may be represented by this mechanism as we will explore in a later section. In a similar vein, identifier subscripting is independent of the *sequence* concept. If an element in a sequence is subscripted, its subscript need not have any relationship to the order of elements in the sequence.

1.3. Sets and Sequences

Sets and sequences constitute the fundamental database "structures" in ADAMS. If the elements of the set were "tuples", then one would customarily call the set itself a "relation"; if the elements of the sequence were "records", one would call the sequence a "sequential file", or a "stream".

Sets and sequences of elements are characterized by the following properties

(1) A set or sequence is *uniquely identifiable*. While they consist of collections of elements, they are themselves also elements.
(2) All sets and sequences are finite.
(3) All sets are *unordered*.
(4) The elements of a set or sequence must all be of the same *class*, or type. If, for example, several elements of class "Q" were instantiated, any number of them could belong to one, or more, sets or sequences. Elements of different classes, say "Q" and "XX3", can not belong to a common set or sequence.[6]
(5) Sets are *elements* of the class SET. More accurately, they belong to a class of the form "SET of <class>", where <class> denotes the class of its constituent elements. Sequences are *elements* of the class SEQUENCE; they belong to a class of the form "SEQUENCE of <class>", where <class> denotes the class of its constituent elements.
(6) All of the standard set operations (except unary complement) may be performed on ADAMS sets. Specifically, ADAMS recognizes: *union*, *intersection*, *relative complement*, element *insertion*, element *deletion*, set *membership* test, test for *emptiness*, *cardinality*, and *looping* over a set.
ADAMS sequences support *iterators* [GrG83, GrO88] by means of the *get_first* and *get_next* operators. They can be manipulated by *append* element, subsequence

[5] The meaning of a name is often called *metadata.* It may include the class of an element, the various attributes defined on a class, etc.

[6] unless it is a set of elements whose class is a super-class of both.

insertion, and subsequence *deletion* operators.

1.4. Codomains

The concept of a "data value" is purposely left undefined in ADAMS. Data values are strings of bits that other processes manipulate in various fashions. ADAMS will store, and retrieve, such data values on request of a host process, so it needs to know enough about such data values to facilitate their storage and later retrieval — but no more.

A codomain may be regarded as an abstract set of data values. For example, the half open interval [0,10) consisting of all real values x, $0 \leq x < 10.0$, could be a codomain. Conceptually a codomain may be infinite, as is this half open interval, even though it is bounded and finitely describable.

More precisely, in ADAMS, a *codomain* is any regular set that is definable in LEX (or a LEX-like language). A codomain is a "nameable" set, but it cannot itself be an element (in the ADAMS sense) and it cannot be generally manipulated in the manner of the sets in the preceding section. The only set operation that can be performed with a codomain is that of set membership. Given a particular value, it can be determined whether it is a member of the codomain (i.e. accepted) or not. Generally, a codomain can be regarded as the specification of the legal form of a kind of data value.

Codomains must be named.

We close by noting that the ADAMS codomain concept is analogous to that of "domain" in many other database models. And they will be used in a similar manner; for example, the value of any attribute mapping must be an element of a codomain. We choose to call the concept a codomain (rather than domain) for two reasons. First, it is mathematically more correct since an attribute maps from a set of elements (its domain) to a set of values (its codomain). Secondly, we chose to restrict the concept to regular sets. The different terminology emphasizes this restriction.

1.5. Attributes and Maps

Attributes and maps are both functions, that is, they are singled valued. The only distinction is the nature of their respective image sets. An *attribute* is a function mapping elements into a codomain; that is, the image of an attribute is a value. In contrast, a *map* is a function mapping elements into a class; its image is another ADAMS element.

In DAPLEX [Shi81] these concepts are called *property* and *function* respectively. In ORION [KBC87] the only mechanism for associating two different objects is an *attribute* concept, because there is no clear distinction between "objects" and "values", such as we have made. However, "primitive objects" behave similarly to our "values". Relational database languages, (c.f. [Cha76, Cod70, Kim79, Mai83, StW83]) use the term *attribute* identically (except that we call the image space a codomain). The relational model has no analog to a map; such inter-class relationships are implemented by the join operator.

Let *a* denote an arbitrary attribute, and let *x* denote any element in the database on which *a* is defined, then the ADAMS expression *x.a* denotes some value in the codomain of *a*. Note that every codomain has a default *udf* (undefined) value; so the expression *x.a* itself is well defined whether or not a specific attribute value for *a* has been assigned for *x*.[7] The ADAMS statement

x.a ← | count int |

[7] The default *udf* value can be user redefined. Users can also define a *ukn* (unknown) value to more precisely indicate the nature of missing data values.

assigns the value of the host language[8] integer variable *count* as the image of *x.a* provided that the value of *count* is actually in the codomain of the attribute *a*. The statement

| count int | ← x.a

will assign the current image of *x.a* to the host language variable *count*, on which further processing can be performed. These ways of referencing codomain values correspond to *putDomVal* and *getDomVal* in [CAD87].

Attributes are a kind of element. Consequently, we may create sets of attributes and apply set operations to them. This capability is important in both the relational approach to databases and the handling of array data.

As we noted at the beginning of this section, a *map* is a function which maps an element into a *class of elements*. Thus application of a map to an ADAMS element yields another ADAMS element instead of a codomain value produced by an attribute function; otherwise they behave just like attributes. Some find it easier to think of a map as an element pointer value.

Frequently, the class of elements which constitute the image of a map will be a restriction of the class SET. This introduces the ability to construct one-to-many and many-to-many maps, which we explore more fully in section 4.4.

1.6. Concise Summary of Primitive Concepts

All of the fundamental properties of ADAMS database concepts are outlined here.

(1) **element:**
- a) uniquely identified, and optionally nameable;
- b) must be typed with some class membership;
- c) pre-defined classes are:
  - 1) SET of ...
  - 2) SEQUENCE of ...
  - 3) ATTRIBUTE
  - 4) MAP
- d) may be a member of multiple sets or sequences.

(2) **codomain:**
- a) regular set;
- b) must be named, but not manipulable;
- c) constituent values are not uniquely identified.

(3) **set:**
- a) uniquely identified, and nameable;
- b) must belong to a SET class;
- c) all elements must be of the same class;
- d) can be argument to set operations;
- e) can be used with iterators.

(4) **sequence:**
- a) uniquely identified, and nameable;
- b) must belong to a SEQUENCE class;
- c) all elements must be of the same class;

---

[8] To distinguish host language variables and their types we delimit them by | ... |. This would be unnecessary if we were willing to parse host language declarations as well as ADAMS statements in the preprocessor.

d) can be used with iterators.

(5) **attribute:**
a) function from an element to a codomain;
b) uniquely identified, and normally named.

(6) **map:**
a) function from an element to an class (often a SET);
b) uniquely identified, and normally named.

The following sections describe syntactic constructs that assume these properties.

2. ADAMS Data Definition Syntax

The preceding sections described the kinds of basic data elements that are embraced by the ADAMS language. In this section we explore many of the linguistic constructs that we have adopted to introduce these data constructs into computer programs. As noted earlier, ADAMS is not concerned with data processing. Its entire focus in on the storage and retrieval of "objects" (elements and their associated values) and sets and sequences of these elements. Consequently, the language emphasizes constructs which can be used to describe the structure of such elements. In the preceding sections we have described the primitive elements which ADAMS understands. They are: *values*, or bit strings that have meaning to a host language, and *elements*, which are essentially distinct aggregations of values, belonging to a *class* of all such elements; but which may also be distinct objects such as *functions*, *sets*, *sequences*.

But it is inconvenient to repeatedly develop database applications in terms of these very primitive concepts. It is slow and error prone. In the visualization and subsequent design of databases we often think in terms of various kinds of composite structures such as, *records* and *files, relations* and *relationships.* These are the kinds of higher level concepts that are primitive in most database design. ADAMS can bootstrap itself up, in a manner analogous to standard object-oriented programming languages [Str87, 81] [9], so that the database designer can name these more complex data constructs and treat them as basic elements in his, or her, database view. But to do this we must develop additional linguistic and meta-linguistic concepts.

2.1. Class Description

In ADAMS, sets, elements, codomains, maps, and attributes denote the kinds of things that can be accessed and manipulated. In a sense they represent the *nouns* of the language. The class to which they belong, or more properly its name, is a *common noun* denoting in a generic sense the characteristics and properties of all such things. We may speak of the class PERSON, that is, the collection of all objects which are people. But we may also speak of a particular person, say *James French*, who is an instance of the class. Specific instances constitute *proper nouns* in the ADAMS language.

For pedagogical purposes, whenever a word in this paper denotes an entire class, it will be set in UPPER CASE. Whenever the word identifies a specific instance, it will be set in lower case characters. Reserved words have been emboldened for emphasis. The following three

[9] However, we again emphasize that ADAMS itself is not properly an object-oriented database system in the manner of EXODUS [CDV88] or ORION [KGB90].

ADAMS statements[10] follow this convention.

```
Q        isa CLASS

Q_SET    isa SET, of Q elements

q_set    instantiates_a Q_SET
```

Q denotes a class (which is arbitrary). *Q_SET* denotes the class of all possible sets whose elements are of class *Q*. *q_set* denotes a particular instantiated set of some such elements. ADAMS does not enforce this upper-lower case distinction, but we find its consistent use enhances reader comprehension.

The first and last of the statements above illustrate the most basic <class_declaration> and <element_instantiation> forms. The second statement has an additional <set_elements_clause> which further describes the type of elements to be found in a *Q_SET*. Such clauses may be separated by optional commas, as shown above.[11]

2.1.1. Associating Attributes and Maps with a Class

Attribute functions are defined on elements. That is to say, some classes (or kinds) of elements have some attributes defined on them. These attributes are associated with the class of elements by the **having** construct. Consider the following sequence of ADAMS statements.

```
REAL_ATTRIBUTE     isa ATTRIBUTE, with image REALS
INT_ATTRIBUTE      isa ATTRIBUTE, with image INTEGERS
STRING_ATTRIBUTE   isa ATTRIBUTE, with image STRINGS

a1      instantiates_a REAL_ATTRIBUTE
a2      instantiates_a INT_ATTRIBUTE
a3      instantiates_a STRING_ATTRIBUTE

Q       isa CLASS, having { a1, a2, a3}

Q_SET   isa SET, of Q elements
```

Attribute functions are elements, and all elements must belong to a class. The class of any attribute is a sub-class of the pre-defined class ATTRIBUTE, where the image codomain is specified. In the example above, we have created three generic attribute classes — *REAL_ATTRIBUTE*, *INT_ATTRIBUTE*, and *STRING_ATTRIBUTE*. $a1$ is declared to be (the name of) a specific *REAL_ATTRIBUTE*, in much the same way that f might be declared to be a "real valued function", or g might be declared to be a "continuous, complex valued function". Thus this sequence has declared that any element in any set of class *Q_SET* will be of class *Q* and each element will

---

[10] To the ADAMS language interpreter, all reserved words are only tokens. For presentation, and initial exposure to the language, long unwieldy tokens such as **instantiates_a** carry valuable mnemonic meaning. Practiced coders soon use equivalent abbreviated tokens, such as **::=.**

[11] Entire ADAMS statements are delimited by << ... >>. These delimiters serve as a convenient way for our preprocessor to separate ADAMS statements from those of the host language, and will be omitted in this paper.

An ADAMS statement may be continued on multiple lines for readability.

All commas are treated as white space; and may occur anywhere that a blank, tab, or newline can appear.

have the three attributes { *a* 1, *a* 2, *a* 3 } defined on them.

For a different and slightly more complex example consider the following sequence in which we use more traditional file processing terms. In spite of the different terminology, the underlying structures of the classes *Q_SET* and *PERSONNEL_FILE*, are very similar.

```
DATA_FIELDS  isa SET, of ATTRIBUTE elements

pdata        instantiates_a DATA_FIELDS,
             consisting of { p_nbr, name, age, dept, salary }

PERSONNEL    isa CLASS, having attributes = pdata

PERSONNEL_FILE    isa SET, of PERSONNEL elements,
             having attributes = { date_last_mod }
```

In this example, we have assumed that the attribute functions, *p_nbr*,...,*salary*, have been previously instantiated. Notice that *DATA_FIELDS* denotes any set of attributes, and that *pdata* is one such set. Normally, when an element of type SET is instantiated, it is instantiated as an empty set, ∅. But in the code above, we used the **consisting of** clause to initialize *pdata* to an enumerated set of attribute elements. An instantiated set of the class *PERSONNEL_FILE*, we might call a *personnel_file*; another might be called the *manager_file*. Elements of the file are of class *PERSONNEL*; which in traditional unit-record terminology would be called personnel records. The data fields of each record are *p_nbr, name, age, dept,* and *salary.* While these attributes (fields) are associated with the elements (records) of the set (file), there is also an attribute (field) associated with the set (file) as a whole; that is, *date_last_mod,* the date the file was last modified.

The **having** construct can only associate extant sets with a class. In the preceding example, the set *pdata* was instantiated and named before defining the class *PERSONNEL*. The enumerated set of defined attributes { *date_last_mod* } was implicitly created in the definition of the class PERSONNEL_FILE. The set *pdata* (since it has a name) is independently modifiable. We can change the definition of what it means to be a *PERSONNEL* by simply manipulating the set *pdata*. Any change to the set *pdata* will not only change the meaning of *PERSONNEL*; it will also be dynamically reflected in every instantiated element of the class. By deliberately treating attributes as elements that can be included in, or deleted from, sets like any other data element, the ADAMS language introduces provision for dynamic schema modification.[12] Because it cannot be independently referenced, the unnamed set { *date_last_mod* } can not be so changed.

The issue of naming elements, classes, and codomains, and of what names are known to what processes, is important in ADAMS.

Note that the class *PERSONNEL* has used the name *attributes* to be an additional *synonym* for its associated set of attributes. In conjunction with the association operator (section 2.2), these synonyms may be used to provide a mechanism by which different users can be provided with different "views" of the elements in a database. The use of such synonymous designators is completely optional.

[12] To support this linguistic feature, our ADAMS system [PFG89] implements attributes literally as functions, not fields in a static structure. Other implementations could be equally effective.

2.1.2. Attribute Denotation Operator

In the preceding section, we instantiated the attributes $a1$, $a2$, and $a3$ which belong to the generic attribute classes *REAL_ATTRIBUTE*, *INT_ATTRIBUTE*, and *STRING_ATTRIBUTE* respectively. The names $a1$, $a2$, and $a3$ denote the attribute functions themselves. Let q be an instantiated element in the class Q. The **having** clause in the class declaration of Q has asserted that $a1$, $a2$, and $a3$ are defined on any element of the class; consequently they are defined on q. In ADAMS, the corresponding image values are denoted by the *expressions*

q.a1, q.a2, and q.a3 .[13]

Map images are treated in the same fashion. If m is an instantiated MAP whose image class is Q, and if x denotes an element belonging to a class on which m is defined, then

x.m

denotes that element of Q which is currently the image of x under m.

Use of the postfix "dot" notation to express functional evaluation, instead of the more traditional prefix notation, such as $a1(q)$ or $m(x)$, simplifies the expression of functional composition. The expression

x.m.a1

is well-formed, and denotes the current image value of $a1$ applied to the element of the class Q which is the current image of x under m. Thus, ADAMS enables "navigational", or "pointer chasing", expressions which adherents of the relational model find abhorrent [Ste90]. But, the design goals of ADAMS have been to implement a variety of database models. ADAMS users can choose abstract database models which employ, or avoid, such constructs.

2.1.3. Predicates

A *predicate* is any expression, P, that evaluates to true or false. For example, $(\forall\ s \in q\_set)\ [\ class\_of\ (s) = Q\ ]$ is a predicate. It would be expressed in ADAMS as

(**all** s **in** q_set) [s.class_of = Q]

Predicates can also involve attribute values, as shown in the next example. Let *set_1* and *set_2* be two sets whose elements belong to classes on which the attribute a is defined (they need not be the same class). If x denotes an element in *set_1*, then $x.a$ denotes a specific value in codomain of a. The predicate

(**all** x **in** set_1)(**exists** y **in** set_2) [x.a = y.a]

will evaluate to true if and only if for every element currently in *set_1* there is at least one element in *set_2* having the same a attribute value. Of course, any change in the composition of either *set_1* or *set_2*, or of their assigned attribute values, may change the truth value of this expression.

Throughout ADAMS we assume that the values of any codomain can be ordered, and that a predicate expression denotes a decidable process.

The first assumption can be assumed, if necessary, as a byproduct of the binary encoding scheme. The second assumption follows largely as a consequence of our requiring codomains to be regular sets.

---

[13] Expressions such as $q.a1$ only denote the image of $a1$ on q; it need not automatically access that image value, in the sense of assigning its value to a variable of the host language.

2.1.4. Class Restriction

It is common, in natural discourse, to define a new class of objects by further restricting an existing class. For example:

FISH **isa** ANIMAL, **provided** (**all** x **in** FISH) [x.swim_in_water]

That is, the class of *FISH* is a subclass of the class of *ANIMALS* which is further restricted by the requirement that any member, x, of the class must "swim_in_water". Predicate restrictions are denoted by the clause

provided <predicate_expr>.

The predicate must be true in order for the element to have class membership. Predicates can be composed with the usual boolean operators, *and*, *or*, and *not*.

Predicate expressions of the form $(\forall x \in <set\_class>)$ [$x.class\_of = <element\_type>$], which restrict the <set_class> to a single specific class of elements, are so ubiquitous that we have condensed them to a single set element clause

of <element_class> **elements**.

Provided clauses and **set element** clauses restrict the kinds of elements that can belong to a class. If X is a restriction of Y, which is in turn a restriction of Z, then X "inherits" the properties of both Y and Z. To be a member of X, x must satisfy $predicate_X$ as well as $predicate_Y$ and $predicate_Z$.

2.1.5. Name Definition

ADAMS uses two basic language constructs to enlarge its vocabulary.

<class_name> **isa** <class> [**and** <class>]*
having <associated attributes/maps>
provided <restrictions and/or qualifications>
of <element_class> **elements**

<instance_name> **instantiates_a** <class>
consisting of <enumerated set>

The first defines (or names) an abstract "class" of things.[14] The second names a specific instance (thing or object) within a class. The **isa** construct is used to define new classes. It is not a specific inheritance operator as in [ACO85, Car84] or [BuA86]. Inheritance is, however, implied because <*class_name*> is a sub-class of <*class*> and thus inherits all of the properties of <*class*> (or of all <class>es in its super class list.

The <*class_name*> or <*instance_name*>, together with its corresponding definition is automatically entered into the users local dictionary. It is now part of the user's own database language. The name and its definition may be later exported to more global dictionaries to become part of a larger database group, or system, vernacular.

[14] The **having, provided,** and **consisting of** clauses are optional in any class definition, or element instantiation statement, as are other clauses such as the **scope** clause which governs the "visibility" of the defined "name".

2.1.6. Parameterized Class Definition

In section 2.1.1, we illustrated the creation of a class *PERSONNEL_FILE*. First, the class *DATA_FIELDS* denoting any set of attributes was declared. Then a specific set of attributes *pdata* was constructed by enumeration. The class *PERSONNEL* denoting a class of elements (or records) all of which have the associated attributes in *pdata* was declared. And finally, the class *PERSONNEL_FILE* was declared as any set of elements (records) belonging to *PERSONNEL*. At best, such a definition mechanism is tedious. At worst, this mechanism (1) is error-prone, (2) clutters up the dictionary with a lot of class names that may never again be referenced, and (3) defines only a single, rigidly constrained kind of "file". Let us now explore how one could define a generic sequential file in ADAMS; this time using the sequence construct.

Consider the following ADAMS declarations

```
DATA_FIELDS  isa  SET,  of  ATTRIBUTE  elements

$Z_RECORD    isa  CLASS,
                      having  attributes = $Z

FILE_OF_$Z   isa  SEQUENCE,  of  $Z_RECORD  elements,
```

In the second ADAMS statement, *Z* is a macro parameter. It must be instantiated as in:

```
pdata          instantiates_a  DATA_FIELDS,
               consisting of  { p_nbr, name, age, dept, salary }

personnel_file instantiates_a  FILE_OF_pdata
```

or

```
manager_file   instantiates_a  FILE_OF_pdata
```

Now the class name is literally, *FILE_OF_pdata* and any element of the file is a *pdata_RECORD*. The instantiation segment *pdata* becomes part of the name. The use of macro instantiations provides an economical way of declaring a large number of similar classes. For example, with just the two basic classes *DATA_FIELDS* and *FILE_OF_$Z*, one can define an indefinite number of simple files.

There is an obvious potential problem with the declaration of *FILE* given above. Its macro parameter could be instantiated with any name, not necessarily the name of a set of attributes. ADAMS employs an **only if** clause to test for consistency in such macro substitutions, as in

```
$Z_RECORD  isa  CLASS
               having  attributes = $Z
               only if  (all  x  in  $Z) [ x.class_of = ATTRIBUTE ]
```

The **provided** and **only if** clauses are similar, yet different. The predicate of a **only if** clause is tested only once — on declaration. If it is not satisfied the declaration is aborted. A **provided** clause becomes part of the persistent declaration. Its predicate may be tested repeatedly at run-time whenever elements are instantiated or manipulated to insure database consistency. If the predicate is not satisfied the statement fails[15] and the process is aborted.

[15] The possibility of an ADAMS statement *failing* is an important concept that is not found in many programming languages. We have borrowed the concept from SNOBOL [GPP68].

All database languages must make some provision for statement, and/or transaction, failure if one is to enforce concurrency criteria, such as serializability, or consistency constraints as above.

2.2. Designators

A *name* uniquely designates something. In ADAMS, a name is a *literal* identifier, just as the literal '-5.63' uniquely designates the corresponding real value. In ADAMS, names are literal symbol strings that are accessible to the external world (that is to a host language program) through the dictionary. Most names simply denote specific database objects, such as a file, a relation, or a set. Given the name of the object, interrogation of the dictionary will return all the information a process needs to use the object. Other names denote element classes, codomains, and attribute and map functions. The purpose of many ADAMS statements is to create and/or interpret dictionary entries. All sets, classes, codomains, attributes, and maps are *nameable*. Codomain values are not nameable — in the sense that literals denoting such values are never entered into the dictionary.

A *designator* is any expression that denotes one, or more, things in an ADAMS database. Every name is a designator. But elements may be designated by mechanisms other than literal naming.

For example, in the ADAMS set looping construct

for_each x **in** q_set **do** ...

x successively designates individual elements of the named set *q_set*. Such variable designators are called *ADAMS variables* to distinguish them from other variables declared in the host language, and must be declared before use by a statement of the form

ADAMS_var x, y, z

These ADAMS variables introduce the capability of designating elements of the database without assigning literal names to them all, in the same manner that the variables of a program support the manipulation of numeric and character values without treating each as a literal constant. Most of the elements of any database are instantiated without ever being literally named, as in

x **instantiates_a** Q
insert x **into** q_set

where *x* is a variable element designator.

Variable designators are clearly central to the ADAMS interface. But, possibly more important, are *set designators*. In section 2.1.1, we declared classes with associated sets of attributes using a **having** clause; and we initialized newly instantiated sets by means of the **consisting of** clause. In both cases, the association set and initializing set were denoted by *enumerated sets*. Standard set notation is used to denote enumerated sets. That is, the set is denoted by a curly brace followed by an enumerated list of its constituent elements and a closing curly brace, as in

{ <element_designator> [, <element_designator>] }.

The <element_designator>s comprising the enumeration list are frequently literal identifiers; but need not be.

In addition to enumeration, set designators can be created using predicates. For example, the expression

{ x **in** q_set | x.a3 = 'Computer Science' }

designates the subset of *q_set* comprised of those elements *x* whose *a*3 attribute have value

'Computer Science', if any.[16] In the predicate

x.a3 = 'Computer Science'

the ADAMS variable *x* is free. In the overall set designating expression, the variable has been bound to designate an element in the instantiated set *q_set*. There can be no free variables in a set designator.[17]

Set designators of the form shown above, are called *retrieval sets*. They represent the most important way that elements of interest are retrieved in ADAMS. They can be quite complex. For example,

```
z     instantiates_a Q_SET
      consisting of { x in q_set | x.a3 = 'Computer Science' and
                                   (exists y in r_set) [ x.a1 > y.a1 ] }
```

creates a new set *z* consisting of those elements of *q_set* which not only have *a*3 value equal 'Computer Science', but also have an *a*1 value that is greater than the *a*1 value of at least one element in *r_set*.

The *association operator, denoted by* →, is used to denote a set that has been associated with an ADAMS class. As an example, consider the second example of 2.1.1, in which we defined a *PERSONNEL_FILE*, and suppose that a user process is given only the name of *personnel_file*, a specific object of that class. The following code could be used to display all values contained in *personnel_file*.

```
for_each x in personnel_file do
        for_each a in x.class_of→attributes do
                write (x.a)
        writeln
writeln
for_each a in personnel_file.class_of→attributes do
        write (a.name_of,  ' = ',  personnel_file.a)
```

Note that in this example the term *attributes* does not uniquely designate any set. The two separate set designators *x.class_of →attributes* and *PERSONNEL_FILE →attributes* do, however, designate unique sets of attributes. Assuming some reasonable implementation of the *write* operator, we would expect this latter code to generate

```
11111   Smith      36   sales        23,000
22222   Johnson    43   engineering  33,500
33333   Lefler     22   sales        19,000

date_last_mod = 7/14/87
```

In mathematical notation, we commonly use literal identifiers (or names) such as *x* or *f* to denote elements or functions of interest. An identifier, such as *x*, can denote anything, although the nature of the thing being designated is usually understood, either by formal definition or by context. When a mathematician exhausts his supply of convenient identifiers, he frequently

---

[16] The vertical bar | is typically read as "such that".

[17] Because all variables are bound to existing sets, evaluation of these expressions is "safe", in the sense of [Mai83, p.247].

begins subscripting them, as in x_1, x_2, or G_α, $X_{\beta,k}$, etc.[18] The important aspect is that subscripting is purely a linguistic convention designed to provide a larger pool of available identifiers; it does not necessarily imply an array structure.[19] ADAMS introduces subscripting for precisely the same reason — to expand the space of available identifiers. We add one additional caveat. *Only instantiated element identifiers can be subscripted.* One cannot subscript class, or codomain names.

An element instantiation of the form

x[*, *, *] **instantiates_a** <class>

indicates that any triply subscripted occurrence of the identifier x, such as

x[n, 13, 0]

will denote an instantiated element in <class>, and may be used in ADAMS expression in the same manner as any other designator of elements in the <class>. The subscripts must be integer; but there is no *a priori* upper, or lower, bound specified in the instantiation statement.[20] In effect, all possible subscript combinations are implicitly instantiated.

This last assertion emphasizes a very important distinction between the ADAMS approach and a more standard object-oriented approach. Typically, in object-oriented languages, when an object is instantiated some memory resident object is actually created. In ADAMS, many elements have only a logical existence; a *uid* is assigned to the element which may appear in functions, or which may appear in sets or sequences. No actual storage need be reserved, to which the *uid* is some kind of pointer. Consequently, instantiation implies only a logical creation — that an element *uid* has, or could be, assigned to this element designator.

3. Implementation Examples

ADAMS is intended to be a basic tool with which many database models may be implemented. In this section we illustrate how one may implement

relational,
hierarchical, and
network

databases using ADAMS primitives. One purpose is to demonstrate the general utility of ADAMS concepts. A second purpose is to provide an abundance of ADAMS examples in juxtaposition with known database organizations.

---

[18] When the mathematician exhausts his convenient subscripts, he often turns to superscripting as well, as in $x^k_{i,j}$. ADAMS only supports subscripting!

[19] Of course, subscripting is a convenient mechanism that can be used in array representation. One can use subscripting in ADAMS as well to denote array structures.

[20] This is an important variant on earlier ADAMS syntax, in which subscripts were drawn from arbitrary *subscript pools*.

On some ADAMS implementations, subscripts have been restricted to non-negative integers, in which case [0, 0, ... , 0] is an implicit lower bound.

3.1. Implementation of a Relational Model

During the last few years, the majority of formal database theory has been couched in terms of the relational model. It has proven itself to be a flexible context for expressing a wide range of database queries and operations, particularly those occurring in business applications. Because of this flexibility, the relational model is the dominant database model, with many available commercial systems and many running applications.

The goal of this section is to show that the basic ADAMS constructs are sufficient to support the relational approach to database design. At the same time we can illustrate use of many ADAMS syntactical constructs in the context of a fairly specific, concrete application. Note, however, that the relational model is a *simple* database model. For example, it requires only the basic concepts of *element, set, attribute* and *codomain.* There is no equivalent of either the *map* or *sequence* concepts. It has been this very simplicity that has rendered it amenable to formal analysis (e.g. [Mai83, Ull82]), and has led to its wide spread use in practice.

We will demonstrate this capability primarily by developing a running example. Recall the hackneyed, but comfortably familiar "Suppliers-Supplies-Parts" database that can be found in any undergraduate text. Its relational implementation consists of three relations *supplier, supplies*, and *part* which have the following schema:

supplier (s_nbr, s_name, s_city, s_zip)

supplies (s_nbr, p_nbr)

part (p_nbr, description, unit_price)

Figure 3-1

We assume that *s_nbr* and *p_nbr* are keys of the *supplier* and *part* relations respectively.

In the following subsections we will illustrate the ways in which the primitive ADAMS concepts can be used to develop this application.

3.1.1. Schema and Relations

We begin by defining ADAMS classes corresponding to the concepts of relational *schema* and general *relations* (with schema) using the **isa** operator to create and name such generic classes.

```
SCHEMA        isa SET, of ATTRIBUTE elements

$Z_TUPLE      isa CLASS,
                    having attributes = $Z,
                    only if $Z.class_of = SCHEMA
$Z_RELATION   isa SET, of $Z_TUPLE elements
```

Now *SCHEMA* denotes a restricted kind of set consisting only of elements of type ATTRIBUTE. A *relation* is a set of elements (or tuples), on each of which a specified set of attributes (called the schema of the relation) is defined. Thus to create any relation, there must first exist a specific named set of attributes to serve as its schema. In the definition of RELATION above, the macro parameter Z will be replaced by the name of the set serving as the schema, as described in section 2.1.6. That named set is associated with the class, becoming the schema of the elements of the relation by a **having** clause.

In the next three ADAMS statements, we first create a specific *schema* consisting of three distinct attributes, *a1, a2,* and *a3.* Then we create two different relations *r* and *s* which have

identically the same schema R.

```
R instantiates_a SCHEMA, consisting of { a1, a2, a3 }

r instantiates_a R_RELATION
s instantiates_a R_RELATION
```

r and s denote specific sets of elements, and may be manipulated as such. Since their elements are of the same class, operations such as $r \cup s$ and $r \cap s$ will be well defined.[21] Individual tuples can be instantiated, assigned values, and inserted into relations as in

```
t instantiates_a R_TUPLE
t.a1 ← '5'
t.a2 ← '-3.7'
t.a3 ← 'Smith'
insert t into r
```

We would expect that the classes *SCHEMA*, *$Z_TUPLE*, and *$Z_RELATION* would be so common and so important, that they would be persistently declared with a system-wide status.

3.1.2. A 'Supplier-Parts-Supplies' Database

Let us assume that generic constructs of *SCHEMA* and *RELATION* have been created by declarative statements as shown in the preceding section. The necessary codomains can be be defined by statements of the form[22]

```
string   isa CODOMAIN, consisting of #[a-zA-Z0-9 ]*#
suppno   isa CODOMAIN, consisting of #[0-9]{5}#
partno   isa CODOMAIN, consisting of #B[A-Z]-[0-9]{4}-[A-H]#
zip      isa CODOMAIN
         consisting of #[0-9]{5} | [0-9]{5}-[0-9]{4}#,
         udf = 'e'
money    isa CODOMAIN
         consisting of #$ | + | - | ∈)[0-9]{1,10}(.[0-9]{2} | ∈)#
```

Generic attribute functions into each of these codomains can be defined by a single parameterized

---

[21] The standard set operators, $\cup$, $\cap$, ~, are readily defined over any two sets whose elements are of the same identical class. They can also be defined over sets whose elements belong to different (even non-comparable in the class hierarchy) classes; but care must be taken to correctly determine the class of elements in the resultant set [Pfa88].

[22] Here we have established the codomain *partno* consisting of all 8 symbol strings whose first character is a 'B' followed by a single upper case letter, four digits, a hyphen, and a single upper case letter A through H. Using a COBOL syntax, this codomain could have been described by PICTURE AA9999XA which provides far less information regarding the structure of these data values. The codomain *zip* permits either the old 5 digit zip codes or the newer 9 digit codes with an embedded hyphen. Finally, the *money* codomain allows 1 to 10 "dollar" digits, optionally followed by a decimal point and 2 "cents" digits and/or a preceding dollar sign or plus or minus sign. Notice that this definition does not permit embedded commas.

All domains are assumed to have an appended default *udf* (undefined) element. In the case of *zip* a specific element, the character 'e', was designated as the udf element.

statement of the form:

```
$Z_ATTRIBUTE  isa  ATTRIBUTE,
                with image  $Z
```

With this, one can instantiate the 7 distinct attributes needed for this example by

```
s_nbr          instantiates_a  suppno_ATTRIBUTE
s_name         instantiates_a  string_ATTRIBUTE
s_city         instantiates_a  string_ATTRIBUTE
s_zip          instantiates_a  zip_ATTRIBUTE
p_nbr          instantiates_a  partno_ATTRIBUTE
description    instantiates_a  string_ATTRIBUTE
unit_price     instantiates_a  money_ATTRIBUTE
```

Once the preliminary declarations have been completed, one can actually create the database itself.

```
SUPPLIER   instantiates_a  SCHEMA
                consisting of  { s_nbr, s_name, s_city, s_zip }

supplier   instantiates_a  SUPPLIER_RELATION

PART       instantiates_a  SCHEMA
                consisting of  { p_nbr, description, unit_price }

parts      instantiates_a  PART_RELATION

SUPPLIES   instantiates_a  SCHEMA
                consisting of  { s_nbr, p_nbr }

supplies   instantiates_a  SUPPLIES_RELATION
```

In the first statement, the type *SCHEMA* has been previously defined to be a set of attribute(s). Thus *SUPPLIER* is the name of a specific *schema* instance which consists of precisely the four attributes *s_nbr, s_name, s_city,* and *s_zip.*[23] Attributes could be added to, or deleted, from this schema set at some later time.

The second statement establishes the specific *relation* named *suppliers.* That is, *suppliers* denotes an arbitrary set whose only constraint is that the attributes of *SUPPLIER* (the RELATION macro parameter) are defined on all of its elements. The set *supplier* is empty, since it has just been created and no elements have been explicitly inserted into it.

It should be apparent that the preceding ADAMS statements have (a) defined the basic relational terminology in terms of the ADAMS primitives, and (b) then created an empty supplier-supplies-parts database. The wordiness of these expressions should be an anathema to most database designers and/or application programmers. This is why we regard ADAMS as a moderately low-level interface. Indeed, we would expect to see many more concise forms of expression such as,

[23] Here we violate the ADAMS convention of setting instances in lower case, because in the relational model one customarily sets schema in upper case and relations with that schema in lower case.

schema SUPPLIER **is** { s_nbr, s_name, s_city, s_zip }.

relation suppliers **has_schema** SUPPLIER.

where these more concise forms can be easily implemented in the language itself, as macro substitutions over the language, or as procedures.

3.2. Using Maps

Before illustrating the implementation of either the hierarchical or network models we must develop the ADAMS concept of a *map* more fully. The relational model does not require a map concept, which is at once both its great strength and its weakness. Mapping concepts are unfamiliar to many database users; so a database model which is based on only flat tables is much easier to explain and to visualize [Cod70]. But invariably relationships must be created between data sets; and the relational join is not always the most effective way of implementing the relationship.

An *attribute* function defined on an ADAMS element functionally associates a single codomain value with that element. A *map* defined on an ADAMS element functionally associates a single ADAMS element with that element. That is, attributes are functions from a class of ADAMS elements into codomains; maps are functions from a class of ADAMS elements into another class of ADAMS elements. The image element of a map may be a set element; thus while maps are properly single valued, they can nevertheless be used to represent one-to-many mappings.

A generic class of maps is declared by naming it and designating its image class, as in:

```
Q_MAP isa  MAP,  with image Q
SUBSET_MAP   isa  MAP,  with image  Q_SET
```

A particular map is instantiated from the generic class, in exactly the same manner that attributes are instantiated.

```
f  instantiates_a  Q_MAP
g  instantiates_a  SUBSET_MAP
```

Here, the image under the map f must be a single element in the class Q, and for any element x, $x.f$ must have been previously established with an assignment statement. The image under the g map will be a set of elements in the class Q.

Maps are associated with elements in a class, just as attributes are associated with the elements of a class. In the following example, any element of class P has the two map functions f and g defined on it, as well as the three attributes *a1*, *a2*, and *a3*.

```
P  isa  CLASS
        having  { f,  g } ,
        having  { a1,  a2,  a3 }
```

Assume the ADAMS statement

```
x  instantiates_a  P
```

Now $x.f$ will denote an element from the class Q. Note that we use the same application operator, to apply maps that we use to apply attribute functions to an element. The expression $x.f.a$ denotes the attribute a defined on the Q type element denoted by $x.f$. Or, since g maps to a

Q_SET of *Q* type elements, we could used the fragment of code

```
for_each y in x.g
        write (y.a)
```

to display the *a* attribute values of every element *y* that is the set which is the image of *x* under the *g* map.

As before, we may choose to provide a synonym for either the set of associated maps or attributes as below.

```
P  isa CLASS
       having maps = { f, g } ,
       having attributes = { a1, a2, a3 }
```

In this example, we have used "maps" to be a synonym for the set of associated maps, and "attributes" to be a synonym for the set of associated attributes. This is simply a convention which allows us to access the set of of *all* associated maps (or attributes) using the association operator

x→maps *x→attributes*

without having to remember each function name individually.

3.3. Implementing a Hierarchical Model

One common usage of maps is the implementation of set valued attributes, or repeating groups, that occur in a hierarchical database. The "personnel file" example of section 2.1.3 can be extended to include a salary history with each record in the file as shown below.

```
DATA_FIELDS  isa SET, of ATTRIBUTE elements

master_rec_data  instantiates_a DATA_FIELDS
                 consisting of { p_nbr, name, age, dept }

salary_rec_data  instantiates_a DATA_FIELDS
                 consisting of { b_date, e_date, salary }

SALARY_REC  isa CLASS
            having attributes = salary_rec_data

SALARY_HIST  isa SET, of SALARY_REC elements

SH_MAP  isa MAP, with image SALARY_HIST

salary_history  instantiates_a SH_MAP

P_REC  isa CLASS
       having attributes = master_rec_data,
       having maps = { salary_history }

P_FILE  isa SET, of P_REC elements,
        having attributes = { date_last_mod }
```

This definition is equivalent to the COBOL data definition

```
01 data record is
        02      p_nbr               PICTURE.
        02      name                PICTURE.
        02      age                 PICTURE.
        02      dept                PICTURE.
        02      salary_history      OCCURS n TIMES.
                03      b_date      PICTURE.
                03      e_date      PICTURE.
                03      salary      PICTURE.
```

except that we have omitted the codomain definitions of the individual attributes, *p_nbr, ..., salary* which in COBOL would have been handled by the indicated PICTURE clauses. Also our definition provides for an attribute *date_last_mod* which is associated with the set (or file) as a whole, and not with any particular element (or record) in it.

3.4. Implementation of a Network Model

The network model, as defined by the CODASYL/DBTG report [AAA71, NNN73], explicitly used the concept of a "set", but only as the set of elements which were the image of a one-to-many map. But the term "map" was never used. Instead, every set was "owned" by an element of some other class.[24] That owner element was the pre-image of the map. The CODASYL (or network) model implements a one-to-many map (for each domain element there is a set (of many) elements that its owns). ADAMS implements many-to-one maps. Consequently, ADAMS descriptions must often include the extra class declaration, of a *set* of image elements. After this is done simple network models are easy to describe in an ADAMS syntax, using a construction very similar to the hierarchical decomposition of the preceding section.

The following ADAMS statements will declare a CODASYL/DBTG schema

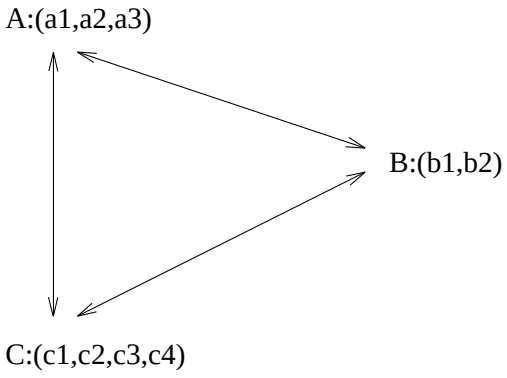

Figure 3-2

where the characteristics of a1, ... ,c4 are defined elsewhere. In this simple network data base, records of type A "own" sets of type B and of type C. Each record of type B also "owns" records of type C.

DATA_FIELDS **isa** SET, **of** ATTRIBUTE **elements**

c_fields **instantiates_a** DATA_FIELDS,

[24] ADAMS uses a FORWARD construct to allow maps to be defined on a class back into itself.

```
consisting of  { c1,  c2,  c3,  c4 }

C_REC  isa  CLASS,  having fields = c_fields

C_SET  isa  SET,  of C_REC elements

C_MAP isa  MAP,  with image C_SET

c_set     instantiates_a  C_MAP

b_fields  instantiates_a  DATA_FIELDS,
          consisting of  { b1,  b2 }

B_REC  isa  CLASS
          having  fields = b_fields,
          having  image_sets = { c_set }

B_SET  isa  SET,  of B_REC elements

B_MAP isa  MAP,  with image  B_SET

b_set     instantiates_a  B_MAP

a_fields  instantiates_a  DATA_FIELDS,
          consisting of  { a1,  a2,  a3 }

A_REC  isa  CLASS
          having  fields = a_fields,
          having  image_sets = { b_set,  c_set }
```

While the preceding sequence is tedious, it need only be executed once, and thereafter retrieved from the persistent dictionary. Unlike COBOL, Pascal, or C, such data descriptions are not included in all subsequent programs — indeed, they *cannot* be repeated!

At this point the definition of the database structures exist, but no elements of type *A_REC*, *B_SET*, or *C_SET* actually exist. Statements of the form

```
a  instantiates_a  A_REC
b  instantiates_a  B_SET
c  instantiates_a  C_SET

a.b_set ← b
a.c_set ← c
```

would create actual sets (elements) of *B_REC*, and *C_REC* elements, that are initially empty; then assign *b* and *c* as the image of *a* under the *b_set* and *c_set*[25] maps respectively. (That is, *a* becomes their owner.) These two sets, *b*, and *c*, need not be the only such sets. Subsequent statements, such as

```
b2  instantiates_a  B_SET
c2  instantiates_a  C_SET
c3  instantiates_a  C_SET
```

[25] Giving a map function the name *b_set* or *c_set* is atrocious! We do it only to reinforce the correspondence with the CODASYL model.

would create additional sets of *B_REC* and *C_REC* records, which (1) may contain common elements and (2) may be involved in set manipulation statements such as

c3 ← c2 **intersect** (a.c_set **union** c3)

where the second operand in the right hand <set_expression> denotes the current image of the element *a* under the *c_set* map.

In the CODASYL/DBTG report, a set can have several owners provided each owner is of a different type. In ADAMS, we relax the latter restriction. ADAMS functions can be defined on distinct sets of the same, or different, class. For example, the map *c_set* is defined on elements in either of the classes, *A_REC* or *B_REC*. And, *c_set* is defined on the elements in both *b* and *b*2, which are sets of the same class, *B_REC*. Consequently, it is hard to define just what is meant by the "inverse image" of an element (or subset) in a set of type C under the "c_set" map.

In the network approach to database design, many-to-many mappings were represented by introducing an extra class of *intersection* records. Basically, a many to many relationship such as "enrolled" between STUDENT and COURSE in the complex network, shown in Figure 3-3,

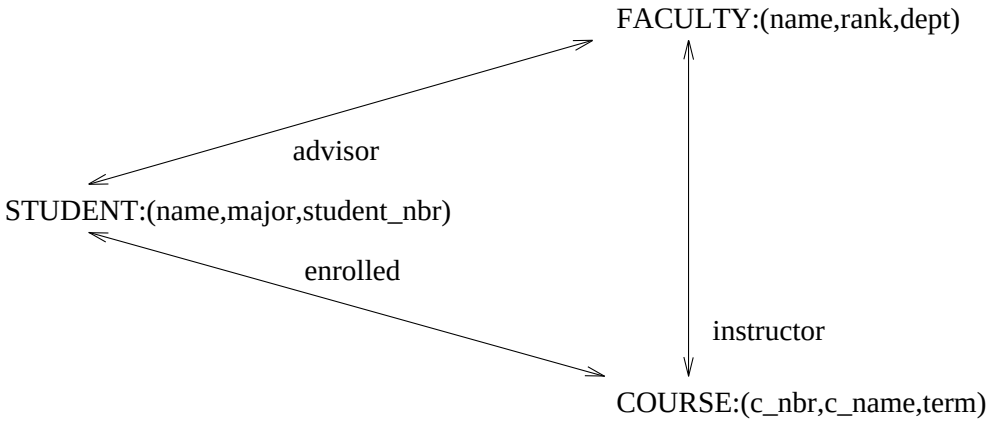

Figure 3-3

must be reduced to two separate one-to-many mappings, "enrolled_in" and "enrollment", as in Figure 3-4, where in a network implementation ENROLLMENT records would be intersection records.

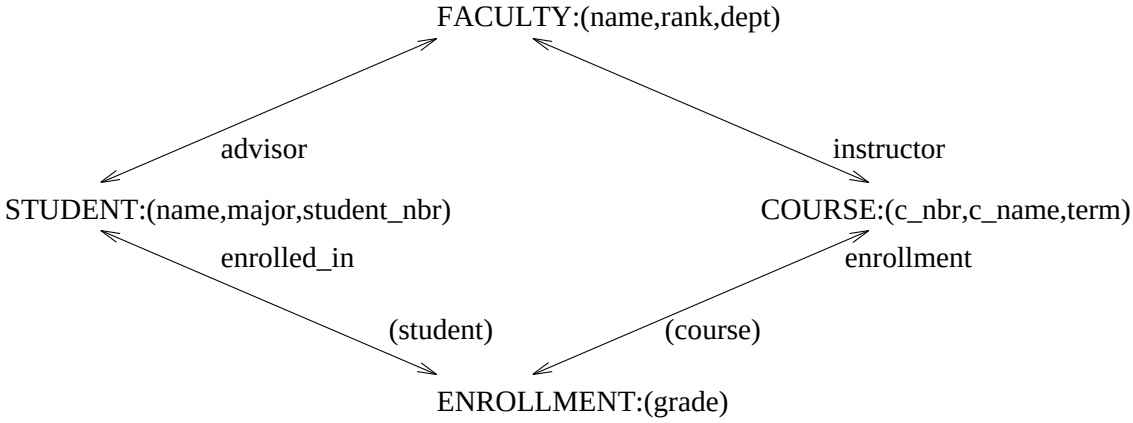

Figure 3-4

In this figure we have named both the one-to-many maps *enrolled_in* and *enrollment*, together

with their many-to-one inverses *student* and *course* (in parentheses). In ADAMS, one always has the option of implementing either the map, its inverse, or both.

Since ADAMS implements its maps as many-to-one functions, it is somewhat easier to avoid the strict CODASYL analog, and implement the inverse maps *student* and *course* on elements of type ENROLLMENT to elements of type STUDENT and COURSE. Then ENROLLMENT elements are truly the *ordered pairs* of a binary relation. Each element of type ENROLLMENT has a *student* "pointer", a *course* "pointer", and a *grade* attribute. A network structure of this nature could be declared by the following ADAMS statements.[26] (This sequence is a fragment of code taken from an actual C program, and various comments have been left in place. The attributes *name*, *rank*, *dept*, *c_nbr*, *c_name*, *term*, *major*, *s_nbr*, *grade*, and *date_last_mod* were previously instantiated as functions with appropriate codomains.)

```
                    /* an initial class declaration required       */
                    /* for map functions, 'advisor', 'instructor'  */
PERSON    isa  CLASS,
             having data_fields = { name,  soc_sec_nbr,  b_date }
PROFESSOR isa  PERSON_REC,
             having fac_data_fields = { rank,  dept }
FACULTY   isa  SET, of FACULTY_REC elements,
             having { date_last_mod }

                                  /* generic map functions */
$1_MAP    isa  MAP,  with image $1

advisor    instantiates_a PROFESSOR_MAP
instructor instantiates_a PROFESSOR_MAP
                          /* class declarations required          */
                          /* for the following map functions      */
STUDENT    isa PERSON,
             having data_fields = { major },
             having maps = { advisor }
STUDENTS   isa SET,  of STUDENT elements
COURSE     isa CLASS,
             having data_fields = { c_nbr,  c_name,  term },
             having maps = { instructor }
COURSES    isa SET,  of COURSE elements
                          /* Final declaration of many-to many    */
                          /* enrollment relationship              */
student    instantiates_a STUDENT_MAP
course     instantiates_a COURSE_MAP

ENROLL_REC  isa CLASS,
              having data_fields = { grade },
              having maps = { student,  course }
ENROLLMENT  isa SET,  of ENROLL_REC elements
```

These statements create the class structure illustrated in Figure 3-4. But, no specific sets of type FACULTY, STUDENT, COURSE, or ENROLLMENT have been created. There could be several in each class. For example, the ADAMS statements

```
courses     instantiates_a COURSES
enrollment  instantiates_a ENROLLMENT
tenured     instantiates_a FACULTY
untenured   instantiates_a FACULTY
graduate    instantiates_a STUDENTS
```

[26] This school database structure has been previously described in [PSF88].

```
undergrad   instantiates_a STUDENTS
```

would create an instantiated collection of data sets which are related in the fashion of Figure 3-5.

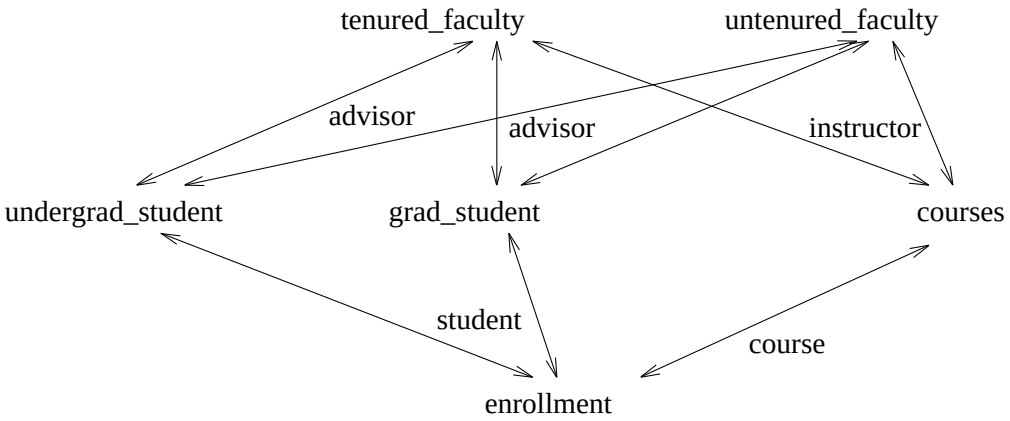

Figure 3-5

All of these instantiated sets are initially empty. Elements can be inserted into the sets with a sequence such as

x **instantiates_a** FACULTY_REC
x.name ← 'Pfaltz'
x.soc_sec_nbr ← '123-45-6789'
x.rank ← 'professor'
x.dept ← 'computer science'
insert x **into** tenured_faculty.

y **instantiates_a** STUDENT_REC
y.name ← 'Smith'
y.soc_sec_nbr ← '012-34-5678'
y.major ← 'computer science'
y.advisor ← x
insert y **into** grad_student

which instantiates a specific faculty element, and a student element with an *advisor* map to the faculty member. Here *x* and *y* would be ADAMS variables; so neither of these two elements are literally named.

In the examples above we have demonstrated that ADAMS can faithfully represent structures arising in the CODASYL network model. From this one might conclude that the flexibility of ADAMS therefore simplifies the task of designing databases. *This is not true.* Instead, the flexibility of ADAMS *complicates* the task of database design. For example, Figure 3-5 is much more complex that that of Figure 3-4 because we have represented multiple instantiations of the class STUDENTS as *undergrad* and *grad*, and multiple instantiations of the class FACULTY as *tenured* and *untenured*. In the network model (and in most object-oriented models) there is only one underlying set — that of the entire class STUDENTS or FACULTY.

As another example, consider the arrows representing maps in Figures 3-2 and 3-4. They are single valued "at one end" and multi-valued "at the other end". What does this mean? What is the map? In the network model, there is no ambiguity because all maps are really one-to-many; in the relational model there is also no ambiguity because all relationships are implicitly many-to-one using key dependencies. We chose to represent the *enrolled* relationship of Figure

3-3 as two many-to-one maps on a set of *enrollment* elements, with the consequence that the multi-valued inverse can be exploited only by retrieval operators, such as

```
{ x  in  courses | (exists  y  in) [ y.student.name = 'Smith'  and  y.course = x ] }
```

to create the set of courses that 'Smith' is *enrolled_in* ; or

```
{ z  in  grad_student | (exists  y  in) [ y.course.nbr = 'CS662'
                                  and  y.course.term = 'S91'  and  y.student = z ] }
```

to retrieve the graduate students enrolled in this semester's CS662 class.

In ADAMS one can easily define "set-valued" maps, such as

```
class_list  instantiates_a  STUDENTS_MAP

COURSE  isa  CLASS
        having  data_fields = { c_nbr, c_name, term }
        having  maps = { instructor, class_list }
```

Now, if we let *y* denote the element representing CS 662 in the term S91, the simple map association operator *y.class_list* suffices to denote the desired set without a retrieval operation. Now we have a direct representation of the many-to-many relationship in Figure 3-3; but not one that is necessarily superior.

The very flexibility with which database relationships can be represented by ADAMS maps makes sound database design essential.

4. Arrays and Sequences

The preceding sections 2 and 3 have primarily emphasized the SET construct of ADAMS because existing database models have largely been concerned with sets of data, whether so named explicitly, or only implicitly implied. In the relational model the schema of a relation is a set of attributes, and the relation itself is a set of tuples. In the hierarchical model, the children of a single parent is a set. Sets are explicitly defined in the network model; they are the image of a one-to-many map — in effect, they are really a functional operator on the owner of the set.

Our emphasis in these sections had been to show that ADAMS constructs can be used to subsume those in a number of standard database languages. In this section, we briefly explore the subscripting and sequence constructs which have not customarily been included in database implementations.[27]

4.1. Arrays

Values structured as arrays are of great importance in scientific applications. As noted in section 2.2, the subscript notation of ADAMS does not necessarily denote an array; any element identifier, including attribute, map, set, and sequence identifiers may be subscripted. But as in

[27] Several implementations of the relational model have provided extensions which allow array attributes. However, there have always been resulting ambiguities, such as "what does it mean to join over an array attribute"?

mathematics, one may use subscript notation to designate elements of an array. In this section we will explore two different ways of representing arrays.

Conventionally, one regards an matrix A as an array of subscripted elements of the form

$$A_{m\times n} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,n} \end{bmatrix}_{m\times n}$$

One intuitively natural way of denoting such an array in the ADAMS syntax is

```
v          instantiates_a  REAL_ATTRIBUTE

MATRIX_ELEMENT   isa   CLASS    having { v }

a[ *, * ]   instantiates_a   MATRIX_ELEMENT
```

so that the subscripted identifiers $a[i,j]$ denote elements in a matrix. But the elements $a[i,j]$ are not themselves real values; they are ADAMS elements. One must apply the attribute v to denote their corresponding values, as in $a[i,j].v$.

While, the code above is intuitively natural, it is not the only possible approach; nor is it necessarily the best. A second representational approach involves subscripting the associated attributes, rather than the "elements" themselves. For example, one could define instead

```
v[ *, * ] instantiates_a  REAL_ATTRIBUTE

3X3_MATRIX  isa CLASS
        having  { v[1,1],  v[1,2],  v[1,3],
                  v[2,1],  v[2,2],  v[2,3],
                  v[3,1],  v[3,2],  v[3,3]   }

a  instantiates_a  3X3_MATRIX
```

In other words, an element of the class $3X3\_MATRIX$ has 9 associated attribute functions, $v[1,1]$ ... $v[3,3]$, defined on it. To denote the value of the $2,3^{th}$ value of the matrix a one would use the expression $a.v[2,3]$. To denote the value of the i,j^{th} value of the matrix a, where i and j are integer variables in the host language, one would use the expression $a.v[i,j]$.

One advantage of this latter representation is that, if inadvertently the expression $a.v[i,j]$ were to be evaluated with either of its subscripts outside of the range [1, 3], the ADAMS statement would $fail$ because the indicated attribute is not defined on a.

A second advantage, is the possibility of generic $m \times n$ matrix declaration as in

```
$M_X_$N_MATRIX  isa CLASS
        having  { v[1..$M,  1..$N] }

b  instantiates_a 3_X_3_MATRIX
```

The attribute v had been declared to be an arbitrarily large, doubly subscripted function. The *range subscript* notation used in the set enumeration of the class declaration above denotes all subscripted attributes within the indicated ranges.

Yet another advantage of the latter representation is the fact that the matrix itself is represented as a single ADAMS element. This makes it easy to associate the entire array with other ADAMS elements by means of a map.

Effective ways of actually implementing these kinds of subscripted representations are described in [PfF90].

4.2. Sequences

Given a provision for subscripting in the database model, there is no actual need for a *sequence* construct. One could define a set of subscripted elements and then manipulate the integer subscripts to impose a sequential order on the set. For that matter, subscripting itself is not strictly required, since one could associate integer attributes *sub_*1, *sub_*2, ..., *sub_n* with an element to emulate subscripting.[28] But subsequent data access and manipulation becomes somewhat convoluted and awkward. The inclusion of subscripted identifiers and sequences in ADAMS makes it a more "natural" database interface, if not literally more "powerful".

The elements of a set are logically unordered; that is, while the looping construct

for_each <ADAMS_var> **in** <set> **do**
.
.

will assign elements to the <ADAMS_var> in some order, it cannot be specified what that order will be. In particular, the order of elements in the <set> need not be the order in which they were inserted.

In section 2.1.6 we used the sequence construct to define generic sequential files. As a second example, assume that the basic elements of a data set correspond to events that must occur, in some order, in a simulation. Each such event element might belong to the class *EVENT*. The set of all events might be instantiated as a set, as in

EVENT_SET **isa** SET **of** EVENT **elements**

all_events **instantiates_a** EVENT_SET

Specific schedules of these events could then be defined by the statements

SCHEDULE **isa** SEQUENCE **of** EVENT **elements**

sched_1 **instantiates_a** SCHEDULE
sched_2 **instantiates_a** SCHEDULE

Event elements could be drawn from the set *all_events* and appended to *sched_*1 and *sched_*2 to reflect different execution schedules, which might then be compared with each other. Notice that, just as an ADAMS element can belong to multiple sets, so can it belong to multiple sequences as well. Moreover, if it were unknown just how many schedules were to be examined, one might choose to declare a subscripted identifier, as in

sched [*] **instantiates_a** SCHEDULE

in which case we have instantiated an indefinite number of schedules.

---

[28] This is the mechanism used by many relational models to represent array data.

A largely unexplored use of sequence construct is the representation of streams of data which may be exploited in parallel processing of large data sets.

5. References

[AAA71] *CODASYL Data Base Task Group Report*, ACM, New York, 1971.

[ACO85] A. Albano, L. Cardelli and R. Orsini, Galileo: A Strongly Typed Interactive Conceptual Lanugage, *Trans. Database Systems 10*,2 (June 1985), 230-260.

[BBK87] F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, FAD, a Powerful and Simple Database Language, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 97-105.

[BuA86] P. Buneman and M. Atkinson, Inheritance and Persistence in Database Programming Languages, *Proc. ACM SIGMOD Conf. 15*,2 (May 1986), 4-15.

[Car84] L. Cardelli, A Semantics of Multiple Inheritance, in *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer Verlag, June 1984, 51-67.

[CDV88] M. J. Carey, D. J. DeWitt and S. L. Vandenberg, A Data Model and Query Language for EXODUS, *Proc. SIGMOD Conf.*, Chicago, IL, June 1988, 413-423.

[Cha76] D. D. Chamberlin, Relational Data-Base Management Systems, *Computing Surveys 8*,1 (Mar. 1976), 43-66.

[Cod70] E. F. Codd, A Relational Model for Large Shared Data Banks, *Comm. of the ACM 13*,6 (June 1970), 377-387.

[CAD87] R. L. Cooper, M. P. Atkinson, A. Dearie and D. Abderrahmane, Constructing Database Systems in a Persistent Environment, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 117-125.

[CoM84] G. Copeland and D. Maier, Making Smalltalk a Database System, *Proc. SIGMOD Conf.*, Boston, June 1984, 316-325.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.

[GPP68] R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL 4 Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1968.

[GrG83] R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[GrO88] R. E. Griswold and J. O'Bagy, SEQUE: A Programming Language for Manipulating Sequences, *Comput. Lang. 13*,1 (Mar. 1988), 13-22.

[Kim79] W. Kim, Relational Database Systems, *Computing Surveys 11*,3 (Sep. 1979), 185-211.

[KBC87] W. Kim, J. Banerjee, H. T. Chou, J. Garza and D. Woelk, Composite Object Support in an Object-Oriented Database System, *Proc. OOPSLA '87*, Oct. 1987, 118-125.

[Kim90] W. Kim, Object-Oriented Databases: Definition and Research Directions, *IEEE Trans. on Knowledge and Data Engineering 2*,3 (Sep. 1990), 327-341.

[KGB90] W. Kim, J. F. Garza, N. Ballou and D. Woelk, Architecture of the ORION Next-Generation Database System, *IEEE Trans. on Knowledge and Data Engineering 2*,1

(Mar. 1990), 109-124.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[NNN73] *CODASYL Data Description Language*, National Bureau of Standards Handbook 113, U.S. Dept. of Commerce, Washington, DC, 1973.

[Pfa88] J. L. Pfaltz, Implementing Set Operators Over a Semantic Hierarchy, IPC TR-88-004, Institute for Parallel Computation, Univ. of Virginia, Aug. 1988.

[PSF88] J. L. Pfaltz, S. H. Son and J. C. French, The ADAMS Interface Language, *Proc. 3th Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988, 1382-1389.

[PFG89] J. L. Pfaltz, J. C. French, A. Grimshaw, S. H. Son, P. Baron, S. Janet, Y. Lin, L. Loyd and R. McElrath, Implementation of the ADAMS Database System, IPC TR-89-010, Institute for Parallel Computation, Univ. of Virginia, Dec. 1989.

[PfF90] J. L. Pfaltz and J. C. French, Implementing Subscripted Identifiers in Scientific Databases, in *Statistical and Scientific Database Management*, Z. Michalewicz (editor), Springer-Verlag, Berlin-Heidelberg-New York, Apr. 1990, 80-91.

[SSE87] A. Sernadas, C. Sernadas and H. Ehrich, Object-Oriented Specification of Databases: An Algebraic Approach, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 107-116.

[Shi81] D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, *Trans. Database Systems 6*,1 (Mar. 1981), 140-173.

[Ste90] M. Stonebraker and et.al, Third-Generation Database System Manifesto, *SIGMOD RECORD 19*,3 (Sep. 1990), 31-44.

[StW83] Q. Stout and P. Woodworth, Relational Databases, *Amer. Math. Monthly 90*,2 (Feb. 1983), 101-118.

[Str87] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, MA, 1987.

[Str88] B. Stroustrup, What is Object-Oriented Programming?, *IEEE Software*, May 1988, 10-2.

[Ull82] J. D. Ullman, *Principles of Database Systems, 2nd Ed.*, Computer Science Press, Rockville, MD, 1982.

[Weg87] P. Wegner, Dimensions of Object-Based Language Design, *Proc. OOPSLA '87*, Oct. 1987, 168-182.

[81] The Smalltalk-80 System, *BYTE*, Aug. 1981, 36-48.

Table of Contents