DOE/ER/25063--T2

DE89 010925

The ADAMS Database Language

John L. Pfaltz, James C. French
Andrew Grimshaw, Sang H. Son
Paul Baron, Stanley Janet, Albert Kim
Cathy Klumpp, Yi Lin, Lindsey Lloyd

IPC-TR-89-002
February 28, 1989

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

DISCLAIMER

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

Abstract

ADAMS provides a mechansim for applications programs, written in many languages, to define and access common persistent databases. The basic constructs are *element, class, set, map, attribute,* and *codomain.* From these the user may define new data structures and new data classes belonging to a semantic hierarchy that supports multiple inheritance.

Table of Contents

Table of Examples

1. Overview

Oh my God! Not another database language. Well, yes and no. The ADAMS language has been created because we perceive a need that is not fulfilled by existing database languages. But ADAMS is not intended to be a complete language by itself. Instead it has been designed to provide a clean database interface for existing programming languages, such as Ada, C, Fortran, and Pascal.

The reasons for undertaking the ADAMS project are described in the following paragraphs

(1) The relational model, which provides the basis of most current database systems has proven itself extremely valuable for the representation the kinds of data used in most business operations. But deficiencies appear if one tries to use it in data fusion kinds of applications. Foremost, is its inability to adequately represent scientific data using *array* configurations. In some systems, there have been *ad hoc* fixes, such as the definition of "array" data types, to circumvent this problem. However, such an approach violates the relational model, for example, one can not join relations over such array attributes.

(2) A characteristic of most database systems, is that the data sets (relations) belong to distinct separate *databases*. Data sets in one database can seldom be used in conjunction with data sets of another database, for fear of violating internal implementation constraints. This effectively fragments an organization's data. All the available data ought to be conceptually accessible by any process, subject only to limitations imposed by security or privacy.

(3) Existing database languages were designed for large centralized processors, with more recent modifications to accommodate very loosely coupled distributed networks of processors. To fully exploit the potential of tightly coupled parallel processing, one needs a language that encourages parallel database access and processing.

(4) Finally, we note the awkward status of read/write statements in traditional programming languages. In many languages, such as Algol and Pascal [JeW75], they are a kind of stepchild which is explicitly disavowed by the parent language In others, only inherently sequential stream I/O is supported. None, with the possible exception of persistent Pascal [BuA86, CAD87], employ a computational model in which the process is coequal with a permanent database from which specific data items are directly accessible.

ADAMS was created in response to these kinds of perceived deficiencies. This report represents the combined design efforts of its authors over a three month period. It builds on several earlier reports, notably [PSF87] which was later presented at the 1988 Hypercube Conference as [PSF88], [PFW88], and [Klu88]. Each of these has presented fragments of ADAMS syntax. But, much of this early syntax has been modified in the light of trial usage, especially of the prototype interpreter described in [Klu88]. The reader is warned to use only this, most recent, version of ADAMS.

1.1. Goals of ADAMS

The overriding goal in designing ADAMS was to create a flexible database system that would actually be used by a large number of applications programmers. This, in turn, translated into a number of more specific goals which are detailed below.

Flexibility: Data comes in many forms, for use in many different applications. For example, one may want to represent

relations,

scientific arrays,
images and topographic data, and
inference networks.

It was our intention that ADAMS should be able to describe at least all of these different data forms, as well as others we had not considered.

Simplicity: One of the strengths of the relational model is its conceptual simplicity. It is relatively easy to learn and to implement. A common problem that arises when older computational forms are extended is that they become quite complex. There are special cases to learn, and more importantly, to implement. An example is Galileo [ACO85], a strongly typed interactive language which embraces many pre-defined special types.

Our goal has been to keep the number of basic constructs to a minimum. To this end, we envision *sets* as the basic aggregation concept.

Embeddability: We would describe a new language as *embedded*, if its constructs are clearly delimited and can be treated as if they were comment statements in the host language. The host language compiler is untouched and host language statements need not be parsed to interpret ADAMS statements. In contrast, a new language is an *extension* if its constructs become integral components of one, or more, of the host language constructs. A language extension requires a much more sophisticated pre-processor or modification of the host language compiler itself.

ADAMS is deliberately designed as an embedded language. A pre-processor converts ADAMS statements into host language statements. There is no modification of the host language itself. For example, host language variables can be used in ADAMS statements, but ADAMS variables may not appear in host language constructs.

Parallelizability: The language paradigm of existing database systems is based on sequential processes running on a single processor. Given a parallel operating environment, one can implement utility processes in parallel as in [DGS88], but there is seldom facilities for a programmer to exploit the inherent possibility of parallel data access at the *applications* level. ADAMS is not specifically a parallel processing language; but since we are implementing it on the Institute's two hypercube configurations it includes fine grained data denotation which permits the application programmer to designate individual subsets of a distributed database.

Portability: A database system must be capable of operating on different kinds of hardware under different operating systems. The ease with which this is accomplished is the traditional sense of "portability". By keeping its basic constructs "simple", ADAMS supports this kind of portability. It is being concurrently implemented in a traditional multiprocessing environment, and in a parallel processing environment.

Another aspect of "portability" is its ability to be used by several different programming languages in the same hardware environment. For this kind of portability a "real" value when read from the persistent database must be converted to a "real" type that is appropriate for the individual language.

Efficiency: This has not been a primary goal; at least we have not sought *efficiency* in the customary sense. For example, we have not optimized storage structures so that block data transfers can be facilitated.

It is our firm belief that the major speed up in data handling will come from the parallel processing of data sets; and that this in turn will be facilitated by flexible storage mechanisms and flexible naming conventions which may be slow by single processor standards. In ADAMS the database implementor is able to make effective use of parallel processors and storage devices. This will be the source of its efficiency.

1.2. General Philosophy

ADAMS is based on what may be called the *entity database* model [Pfa88]. That is, its fundamental units of organization are "entities", or "objects", or as ADAMS calls them "elements"†. Every ADAMS *element* is uniquely identifiable. One may loosely say that ADAMS is "object oriented"; and in a somewhat different context one might be inclined to call its elements, *objects*. The difference between ADAMS and other object-oriented databases is largely one of degree. For example, ADAMS does not hide the logical structure of the data that it represents—instead its primary function is to publically describe a logical structure. (However, much of the fine-grain implementation structure is hidden.) And, although there exist mechanisms for associating methods with instance elements of particular classes, such methods are neither the sole, nor even the primary, interface mechanism as they are in true object-oriented systems.

ADAMS "elements" are the basis for representing the logical structure of the data. Actual stored data values are drawn from user definable *codomains*. It is possible to create sets of elements in ADAMS, but not sets of data values.

Every ADAMS element must belong to a *class*. The class system supports multiple inheritance [Car84]. In this regard, and in its syntax and usage, ADAMS is a *semantic database* system in the sense of [HuK87].

Probably the most important aspect of ADAMS is its treatment of *names*. Although there are many different ways of referencing desired data elements and their values [KhC86], at some fundamental data access depends on the ability to name elements, or sets of elements, in the database. A familiar paradigm is the use of names to identify variables and procedures in traditional programs. However, the scope of these names is always limited to the program itself. The same name can be repeatedly used in different programs. In contrast, the names of elements in a persistent database must themselves be persistent. And they must be unique. This requires a much larger "name space" and much more sophisticated naming conventions than most programmers are accustomed to.

ADAMS employs a segmented hierarchical name space which allows a programmer to both construct private data names as well as shared, common data names. It also supports the indexing of names, an important mechanism for extending a name space, without the usual connotation that the indexed names denote an array structure.

It should be emphasized that the introduction of persistent names introduces a level of complexity that is completely missing in traditional programming languages; but which must be addressed in any treatment of persistent database access.

1.3. Basic Constructs

ADAMS has only five basic constructs: they are *codomain, class, set, attribute,* and *map.*

All computing systems must have a primative (or atomic) level in which the meaning of a sequence of bits is defined by convention. These are *data values*. In ADAMS the conventional meaning of a sequence of bits is known as a *codomain*. For example, one may have a codomain consisting of "real" numbers, or of nine digit social security numbers, or of all strings beginning

† In this report we will use "element", "entity", and less often "object" as synonyms.

with the letter 'T'. In many programming languages, these would be called "data types". In the relational model, they would be called simply "domains". We use our terminology because they actually serve as "codomains" to attribute functions.

The concept of *class* is fundamental to ADAMS. Every nameable entity must belong to a class. A class represents a generic entity—its structure and its properties. All individual entities, or instances, within the class share the same structure and properties. All classes are declared and named by the user, except for the three pre-defined classes *set*, *attribute* and *map* classes.

In most database processing we work with sets of data items, not just single entities, for example, the set of "all computer science students with grade point average greater than 3.2". Such sets must themselves be entities. They belong to a pre-defined class of type *set*.

Functions can be defined on ADAMS entities. They are distinguished according to their image spaces. An *attribute* is a single valued function whose domain consists of entities in one, or more classes, and whose image space, or codomain is a *codomain*. In contrast, a *map* is a single valued function whose domain consists of entities in one, or more classes, and whose image space, or codomain is a *class*.

In other words, the functional value of an attribute function *a* on a particular entity *x*, denoted by *x.a* will be an atomic data value from a codomain, while the functional value of a map, denoted by *x.m* will be another entity, say *y*.

We would re-emphasize that any entity instance belonging to either a user defined *class* or to a user defined *attribute*, *map*, or *set* class can be named. It has an "independent" existence. Specific values in a *codomain* can not be named. They have no independent existence, save as the current value of an attribute function acting on an entity instance.

1.4. ADAMS Statements

Since ADAMS is an embedded language, every ADAMS statement is clearly delimited—just like a comment. We use the delimiters << and >>, but clearly any other set of delimiters could serve as well. Thus the basic ADAMS syntax is:

<ADAMS_statement> ::= <b_delimiter> <statement_body> <e_delimiter>

<b_delimiter> ::= <<

<e_delimiter> ::= >>

The <statement_body> denotes any of 33 ADAMS statements. These statements may be generally grouped into five general types: those declaring generic codomains and classes; those establishing entity instances; those manipulating sets; those accessing elements and data values; and finally, a few miscellaneous statements. We enumerate all of the different ADAMS statement types below. A more detailed expansion of each will be found in the sections indicated to the right of each statement.

<statement_body> ::=	<open_ADAMS_stmt>	1.4
	<codomain_decl_stmt>	2.2
	<subscript_pool__decl_stmt>	2.2
	<add_codomain_method>	2.2
	<attribute_decl_stmt>	3.3
	<attribute_instance_stmt>	3.3
	<map_decl_stmt>	4.2
	<map_instance_stmt>	4.2
	<class_decl_stmt>	5.2.1
	<elem_instance_stmt>	5.2.1
	<remove_element_stmt>	5.2.1

<variable_decl_stmt>	8.2	
<set_decl_stmt>		6.2.1
<set_instance_stmt>		6.2.1
<fetch_statement>		3.2
<store_statement>		3.2
<looping_statement>		6.2.2
<end_loop_statement>		6.2.2
<set_copy_statement>		6.2.2
<set_assign_statement>		6.2.2
<make_empty_stmt>		6.2.2
<insert_statement>		6.2.2
<erase_statement>		6.2.2
<union_statement>		6.2.2
<intersect_statement>		6.2.2
<complement_statement>		6.2.2
<rescope_statement>		9.2
<transaction_statement>		9.2
<abort_trans_statement>		10.2
<end_trans_statement>		10.2
<lock_statement>		10.2
<unlock_statement>		10.2
<close_ADAMS_stmt>		1.4

There is no well-formed ADAMS program, because the *program* concept exists only in the host language. ADAMS simply consists of one or more ADAMS statements embedded in a host language program or procedure. However, any sequence of ADAMS statements must be preceded with an <open_ADAMS_stmt> and eventually terminated with a <close_ADAMS_stmt>. These have the syntactic structure:

<open_ADAMS_stmt> ::= **open_ADAMS** (<job_id>)

<close_ADAMS_stmt> ::= **close_ADAMS** (<job_id>)

These statements open and close, respectively, various ADAMS dictionaries. They need be issued only by the main program executing on any processor. The <job_id> is used to coordinate execution on multiple processors.

Any ADAMS statement can fail for a variety of reasons. The open_ADAMS statement creates a statement status word, called *A$STATUS*, which can, and should be, tested after executing any ADAMS statement. In Fortran programs this is located in labelled common /ADAMS/.

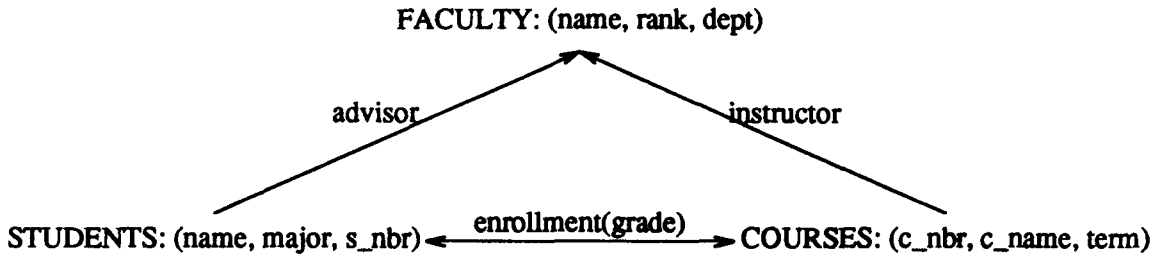
1.5. Running Examples

To provide examples of the ADAMS statements described in the following sections, we will establish two running database examples. The first is designed to illustrate and exercise those features which are used in relational and semantic database models. It was describe in [PSF88] and served as a prototype implementation test vehicle in [Klu88]. The second database will be used to illustrate scientific usage.

A practice used by the ADAMS group, is to capitalize the names of generic sets, such as codomains and classes, and to represent specific entity instances in lower case letters. While this seems to be a valuable convention, it is not an ADAMS rule.

1.5.1. Relational

ADAMS is designed to be more flexible than familiar relational database systems. Nevertheless, relational databases are a fundamental way of structuring information. In Figure 1, we show an entity-relationship diagram for a traditional "students", "faculty", "courses" type database that we will use as a running example to illustrate various ADAMS features.



Entity-Relationship Diagram
Figure 1.

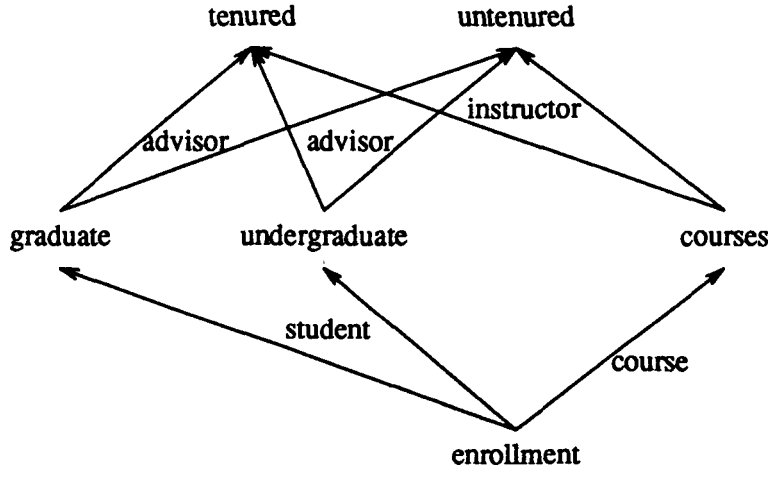
One running example will implement this structure as a 3NF relational database. It will contain the following four relations that one would expect in such an implementation.

Schema	Keys
FACULTY: (fname, rank, dept)	fname
STUDENT: (sname, major, s_nbr, fname)	sname
COURSE: (c_nbr, c_name, term, fname)	c_nbr, term
ENROLL: (sname, c_nbr, term, grade)	sname, c_nbr, term

Here the attribute *fname* in the STUDENT and COURSE schema implements the single valued *advisor* and *instructor* relationships respectively. We will find, however, that it is difficult to capture all aspects of the relational model in an entity based mode. Projection, for example, will not be easy.

1.5.2. Semantic

ADAMS is a database system that is actually based on the semantic model, not the relational model. One consequence of this distinction is that a "relation" is an instance set of "tuple" entities, not a flat table as in Codd's original formulation. Thus FACULTY and STUDENTS denote classes of entities, not specific instances. In Figure 2, one has two different FACULTY "relations" called *tenured* and *untenured*, and two different STUDENT "relations" called *undergrad* and *graduate*. Moreover, the *advisor* and *instructor* relationships are represented as *maps*, not as tuple attributes.



Semantic Database Schema
Figure 2.

1.5.3. Scientific

The running example from the scientific domain is simply a doubly subscripted real array, or matrix. Any programming language can handle such matrices as an aggregate data type. Few database models handle multiply subscripted arrays in a flexible manner. The simplest example will be just a real 3×5 array

$$x = \begin{matrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & x_{3,5} \end{matrix}$$

2. Codomains

2.1. General Description

A data *value* is a finite string of bits which has meaning when interpreted with respect to the conventions of some programming environment. An ADAMS *codomain* is an abstract set of all possible values which can be so interpreted. In this sense, an ADAMS codomain is very similar to the more familiar *data type*, such as "real", "integer", "float", "REAL*4", "boolean", "LOGICAL", etc. The data type "real", used in a Pascal environment on a 8080 chip, specifies how 32 bits should be subdivided so they can be interpreted as the sign, mantissa, and exponent of a real number.

But ADAMS is not concerned with the interpretation of values in a programming environment. It is concerned with the storage of such values in a form which admits later access. As such it is quite concerned with mechanisms for converting (or coercing) values in some storage format into forms that can be interpreted by the accessing process in its own processing environment. It is also concerned with the integrity of the database. Therefore, it is concerned that values stored in the database actually belong to that abstract set specified by the codomain definition.

Consequently, an ADAMS codomain definition has a three-fold purpose:

(1) specification of the form which any value in the codomain will have;

(2) specification of processes to coerce (or convert) values from the storage format used by ADAMS to a form that will be interpretable by the accessing process in its own environment—and, inversely, the conversion of "internal" values back into the ADAMS storage format;

(3) specification of values to be returned (or stored) when an actual value is
   (a) undefined, or
   (b) unknown.

Codomains can be regarded as similar to primative classes in strictly object-oriented languages; however, they are not used to build up higher level classes in the same way.

2.2. Syntax

<codomain_decl_stmt> ::=	<codomain_name> **isa** CODOMAIN <membership_clause> [<access_method_clause >] [<other_method>] [<undefined_clause>] [<unknown_clause>] [<scope_clause>]
<codomain_name> ::=	<actual_name>
<membership_clause> ::=	**consisting of** #<regular_expression># | **validated by** <codomain_method_def>
<access_method_clause> ::=	fetch: <codomain_method_def> store: <codomain_method_def>

† In this syntactic notation, [...] denotes an optional construct; [...]* denotes that it can be repeated indefinitely.

<other_codomain_method> ::=	<method_name>: <codomain_method_definition>
<undefined_clause> ::=	**udf** = <literal_value>
<unknown_clause> ::=	**ukn** = <literal_value>
<literal_value> ::=	' <codomain_value> '
<codomain_method_def> ::=	<extern_def_codomain_method> | <locally_def_codomain_method>
<extern_def_codomain_method> ::=	**EXTERNAL** <name>
<locally_def_codomain_method> ::=	<host_language_proc>
<value_desig> ::=	<element_desig>.<attribute_desig>
<subscript_pool__decl_stmt> ::=	<sequence_name> **denotes_a** SUBSCRIPT POOL **of** <codomain_name> **values** [<consisting_of_clause>]
<sequence_name> ::=	<actual_name>
<add_codomain_method> ::=	**add method to** <name> CODOMAIN <method_name>: <codomain_method_def>

2.3. Semantics

(1) A <codomain_decl> declares a generic set of data values defined in terms of the membership clause; and assigns <codomain_name> as the name of this set. This name is entered into the dictionary, together with its associated information. This definition declares the form that these values will take in ADAMS storage—it does not indicate how they will be represented in any particular computing environment.

(2) To insure database integrity, all codomain values are validated before committing them to *permanent* storage. A value is validated either by comparing it with the <regular_expression> or by invoking the user supplied *boolean* <codomain_method>. This latter can be used to provide user-defined run-time consistency checking, or to circumvent it altogether by having it always return *true*.

(3) ADAMS assumes as its general paradigm that all codomain values are stored as variable length ASCII strings. Therefore, in general, it will be necessary to define <codomain_method>s which convert values between their ADAMS storage format and the corresponding internal computational representation. These format conversion (or coercion) routines are declared in the <access_method_clause>.

Notice that if either a "fetch", or "store" method is declared, then both must be declared.

(4) The presumption that the stored version will be an ASCII string can be changed by providing access methods which convert (or do no conversion) into any user specified form. If no <access_method_clause> is provided, the default assumption is that the internal representation of the value is a *string* (NULL terminated in C), and treated accordingly.

(5) All <codomain_method>s are assumed to be *procedures* with two fixed parameters, the first denoting an internal representation, the second an ADAMS <value_desig>nator. That is, they have the form

```
<name> (int_rep, value_desig)
<type> *int_rep;
char   value_desig;
```

in C. In Fortran, the form would be

```
SUBROUTINE <name> (int_rep, value_desig)
      <type>    int_rep;
      CHAR*<n>   value_desig
```

(6) There exist two pre-defined ADAMS access procedures of the form

```
adams$f (buffer, buf_len, value_desig)
untyped  *buffer;
int      buf_len;
char     *value_desig;
```

and

```
adams$s (buffer, buf_len, value_desig)
untyped       *buffer;
int      buf_len;
char     *value_desig;
```

which f(etch) (or s(tore)) the designated value into (or from) the designated buffer without modification.

(7) A subscript *pool* is a sequential enumeration of codomain values that can be used as subscript values. There is no provision in ADAMS for changing the members of the pool. Only additional values can be added to the pool.

The codomain values of a *pool* must be distinct.

(8) There may be several fetch and store methods associated with a single codomain. For example, a different version of "fetch" will normally be required by each host language used to access the ADAMS database. Similiarly, different hardware architectures may require different conversion routines. Hidden by the ADAMS interpreter is a run-time environment status consisting of (<host_language>, <hardware_system>).

An <add_codomain_method> statement permits the addition of codomain methods, appropriate to new host environments, to an already existing CODOMAIN declaration made in a different environment.

(9) If the <literal value> of either the <undefined_clause> or the <unknown_clause> is not a member of the regular set defined by the <membership_clause>, it is added to the set (finite union).

The **udf** value is returned by ADAMS whenever a <value_desig> has not been defined in ADAMS storage. A **ukn** value must have been previously assigned by the user to <value_desig>.

The default **udf** value is an octal zero, or NULL.

(10) Note that all literal codomain values must be quoted, even if they are numeric. This is in contrast to ADAMS literals which are unquoted.

(11) Codomain and subscript pool names are <actual_name>s, consequently they can be neither subscripted nor parameterized.

2.4. Examples

One would expect most of the commonly used codomains (or types) to be globally declared with SYSTEM scope. Below are samples declaring a REAL codomain for both C and Fortran host languages.

C host language:

```
REAL isa CODOMAIN
        consisting of #( |+|-)[0-9]*.[0-9]*#
        fetch:
                fetch (dest, value_desig)
                float  *dest;
                char   *value_desig;
                       {
                       char   IO_buf[20]

                       adams$f (IO_buf, 20, value_desig);
                       if (*IO_buf != ' ')
                              sscanf (IO_buf, "%f", dest);
                       else
                              *dest = 0.0;
                       }

        store:
                store (source, value_desig)
                float  *source;
                char   *value_desig;
                       {
                       char   IO_buf[20]

                       sprintf (IO_buf, 20, source);
                       if (*IO_buf != ' ')
                              adams$s (IO_buf, 20, value_desig);
                       }
        udf = 0.0
        with scope SYSTEM
```

Fortran host language:

```
REAL isa CODOMAIN
        consisting of #( |+|-)[0-9]*.[0-9]*#
        fetch:
                SUBROUTINE FETCH (DEST, VALUE)
                       REAL   DEST
                       CHAR*30 VALUE
                CHAR*20 BUFFER

                CALL adams$f (BUFFER, 20, VALUE)
                IF (LEN(BUFFER) .GT. 0) THEN
                       READ (BUFFER, '(F20.10)') DEST
                ELSE
                       DEST = 0.0
                ENDIF
                END
```

```
store:
        SUBROUTINE STORE (SOURCE, VALUE)
              REAL    SOURCE
              CHAR*30 VALUE
        CHAR*20 BUFFER

        WRITE (BUFFER, '(F20.10)') SOURCE
        CALL adams$s (BUFFER, 20, VALUE)
        END
udf = 0.0
with scope SYSTEM
```

The subscript pool concept allows the kind of "enumerated subscript" that occurs in Pascal. For instance, if we wanted to subscript ADAMS names with various makes of automobiles we could declare:

```
<<      autos denotes_a SUBSCRIPT POOL of STRING values
              consisting of { 'chevrolet', 'dodge', 'ford', 'plymouth' }
              scope is USER                 >>
```

We can never eliminate 'ford' from the pool or change its spelling; it may have been used to subscript some permanent name. But we can add to a subscript pool as in

```
<<      add 'toyota' to autos POOL          >>
```

Readily the most commonly used subscripts are integer, and we want to declare such a pool of subscripts. We name this pool *Zahlen*, the German word for the natural numbers, that is often used in mathematics. This pool, which we will use repeatedly in our matrix examples, we make a SYSTEM concept.

```
<<      Zahlen denotes_a SUBSCRIPT POOL of INTEGER values,
              scope is SYSTEM               >>
```

This subscript pool is empty. The following bit of C-code inserts the first n non-negative integers in their natural order.

```
        i = 0;
        while (i <= n)
              {
<<            add i to Zahlen POOL          >>
              ++i;
              }
```

2.5. Discussion

The functions of a "codomain" and a subscript "pool" are orthogonal in ADAMS. The former provides values for attribute functions. The latter provides values that can be used to subscript names. The subscript "pool' concept is associated with codomains and included in the section simply because *fetch* and *store* conversion methods must be defined for codomains. This allows subscript operations to piggyback on them.

3. Attributes

3.1. General Description

An ADAMS *attribute* is a single valued function defined on instances of a *class* whose range, or codomain, is a *codomain*. The attribute is itself an ADAMS entity belonging to a class of similar functions that map into the same codomain. For example, the attributes 'age' and 'nbr_of_dependents' might both be instances in a class 'INTEGER_ATTR'.

3.2. Syntax

<attribute_decl_stmt> ::= [**var**] <attr_class_entry> **isa** ATTRIBUTE
with image <codomain_name>
[<association_clause>]
[<restriction_clause>]
[<scope_clause>]

<attr_class_entry> ::= <dict_class_entry>

<attribute_instance_stmt> ::= [**var**] <attr_entry> **denotes_a** <attr_class>

<attr_entry> ::= <dict_instance_entry>

<fetch_statement> ::= **fetch into** <host_variable> **from** <element_desig>.<attr_desig>

<store_statement> ::= **store from** <host_expression> **into** <element_desig>.<attr_desig>

3.3. Semantics

(1) Attributes exist as the functional link between ADAMS entities and their associated data values. What are traditional known as "data values" only exist as attribute images. Thus all "data" must be referenced by the applicative form

<element_desig>.<attr_desig>

(2) The "image_is_clause" is required in all attribute declarations.

(3) The clauses that may appear in an <attribute_decl> may be used in general *class* declarations and are therefore treated in that section.

(4) The representation of attributes is best visualized as an associative "triple", whose components are

(<element_id>, <attribute_id>, <attribute_value>).

Specification of the first two components, as in <element_desig>.<attr_desig> yields the unique third component <attribute_value>. Specification of the second two components will, in general, yield the *set* of <element_id>s that appear as the first component in at least one such triple in ADAMS storage. The syntax for this is discussed in 7.2.

(5) Both designators of the <element_desig>.<attr_desig> of a <fetch_statement> are first evaluated. The designated attribute instance must be defined over the class of the designated instance element. If it is, the corresponding triple (since all attributes are single valued, there can be but one), if any, is accessed for its <attribute_value>. The codomain to which this <data_value> belongs is known—it is the image space of the attribute class to which this instance belongs. Using the "fetch method" of the codomain declaration, this value is converted from its ADAMS storage format into its corresponding computational type and stored in (or assigned to) the <host_variable>.

If no such triple exists in ADAMS storage, then the *undefined* value, **udf** for that codomain

is returned as the value.

(6) The semantics of a <store_statement> are similar. However, in this case the current value of the <host_expression> is converted from its computational format to its ADAMS representation using the "store method" of the codomain. If a triple (<element_desig>, <attr_desig>, <old_value>) already exists, then the <old_value> is replaced by the ADAMS form of the <host_expression>. If no such triple exists (this is the first assignment to this attribute on this entity instance) then a new triple is created.

3.4. Examples

Three distinct steps must be followed before an attribute function can be used to store and access data. First, the codomain must be defined, as in

```
<<    DATE   isa CODOMAIN
             consisting of #[0-9]{2}/[0-9]{2}/88#
             scope is SYSTEM                                    >>
```

Since no access method has been declared, the ASCII string is fetched and delivered as the data value.

Second, a generic class of attributes which map into this codomain must be declared, as in

```
<<    DATE_ATTR    isa ATTRIBUTE   with image DATE,  scope is GROUP     >>
```

And finally, specific attributes (or instances) in this class must be declared, as in

```
<<    b_date         denotes_a DATE_ATTR, scope is USER       >>
<<    date_last_mod  denotes_a DATE_ATTR, scope is USER       >>
```

Now, if *x* is an entity designator (variable, literal name, etc.) and the attributes *b_date* and *date_last_mod* have been defined on the class to which *x* belongs, one can use fetch and store commands of the form:

```
<<    fetch into birth_date from x.b_date            >>
<<    store from today() into x.date_last_mod        >>
```

3.5. Discussion

In earlier versions of ADAMS attributes were designated as either *assigned* (functional value explicitly established by a previous assignment statement) or *computed* (functional value computed on retrieval using other information). Associated with computed attributes was to have been a method, or procedure, for computing the attribute value at retrieval time. The problem is: "where does one define this associated computation method?". It makes no sense to declare it with a generic class of type ATTRIBUTE. Nor does the ADAMS paradigm permit its definition with a particular instance attribute. So it has been eliminated. The effect of a "computed attribute" can be created by defining a method associated with a particular class.

The fact that attributes are themselves ADAMS elements is an important one. Internally, they are represented just like any other entity. Any attribute, or more accurately any class of ATTRIBUTE, may itself have associated attributes or maps (although we have not yet discovered any practical application of this level of generality). However, this has implications in the "dot" notation used to designate data values in ADAMS.

Suppose, for example, that *x* denotes an entity instance in some class on which the attribute instances *f* and *a* are both defined. Suppose further that the attribute *a* is defined on the class of attributes to which *f* belongs. Then, *x.f* and *x.a* designate specific values in the codomains of *f* and *a*, respectively. And *f.a* denotes a value in the codomain of *a*. They are all <value_desig>s. But the expression *x.f.a* is meaningless because the prefix *x.f* is not an <element_desig>.

One implementation approach is to let every attribute instance entry in the dictionary have a pointer to its attribute index structure. (Actually this must be an indirect pointer to allow for subscripts on the <actual_name>.) This index structure is used to access data values given an <element_desig>. A similar inverse index is used to access multiple elements which have a given <data_value>.

The syntax for *fetch* and *store* statements is admittedly cumbersome. A syntax such as

```
<host_variable> <- <element_desig>.<attr_desig>
```

would be much more "natural". These "wordy" fetch and store constructs may have the advantage of emphasizing the nature of these operations; but we should probably consider simplifying the syntax.

4. Maps

4.1. General Description

An ADAMS *map* is a single valued function defined on instances of a *class* whose range, or co_domain, is a *class*. Notice that the only difference between attributes and maps is that the image of the former is always a data value, while the image of the latter is an ADAMS entity, or element. A map is also itself an ADAMS entity that belongs to a class of all similar functions which map into the same class.

4.2. Syntax

<map_decl_stmt> ::= [**var**] <map_class_entry> **isa** MAP
with image <class_name>
[<association_clause>]
[<restriction_clause>]
[<scope_clause>]

<map_class_entry> ::= <dict_class_entry>

<map_instance_stmt> ::= [**var**] <map_entry> **denotes_a** <map_class>

<map_entry> ::= <dict_inst_entry>

4.3. Semantics

(1) A <map_type> is just a dictionary name (possibly parametrized) which belongs to the MAP class. A map instance must belong to a MAP class.

Similarly a <map_name> is just the literal name of a map instance.

(2) The "image_is_clause" is required in all map declarations.

(3) The clauses that may appear in an <map_decl> may be used in general *class* declarations and are therefore treated in that section.

(4) The representation of maps is best visualized as an associative "triple", whose components are

(<element_id>, <map_id>, <map_value>).

Specification of the first two components, as in <element_desig>.<map_desig> yields the unique third component <map_value> which is a unique element identifier. Specification of the second two components will, in general, yield the *set* of <element_id>s that appear as the first component in at least one such triple in ADAMS storage.

4.4. Examples

The following example is based on semantic network of figure 2 in section 1.5.2. Two maps are indicated from the instance sets of *graduate, undergrad,* and *courses* to the instance sets *tenured* and *untenured,* which we will assume comprise entities from the class FACULTY_REC. This class we assume has already been declared. Then the three statements

```
<<    FACULTY_MAP isa MAP with image FACULTY_REC,  scope is USER  >>

<<    advisor denotes_a FACULTY_MAP,     scope is USER      >>
<<    instructor denotes_a FACULTY_MAP,  scope is USER      >>
```

establish these maps. The first ADAMS statement defines the class of FACULTY_MAP functions. It asserts that the image of any such map function will be an entity from the class

FACULTY_REC. *advisor* is then established as one instance of such a map; as is *instructor*.

Note that these map functions have been defined. They have not been associated with entities of type STUDENT_REC or COURSE_REC as yet.

4.5. Discussion

It is much easier to declare generic attribute and map classes using parameterized class declarations, as in Section 8.

Map functions can be implemented in a manner that is virtually identical to that of attributes.

The possibility of having a <restriction_clause> in a map class has been provided, but it is difficult to envision appropriate restrictions at this time. It might be possible to define one-to-one maps by this mechanism.

5. ADAMS Classes

5.1. General Description

An ADAMS *class* is a generic description of a collection of entities with the same, or similar, properties. Generally, the user defines classes that reflect the properties that he (or she) feels characterize the entities in his (or her) database. Since classes can be, and normally are, defined in terms of other classes, a hierarchical class structure arises, which is frequently described by the term *class inheritance*. In fact, the class structure of ADAMS is not really hierarchical since it supports multiple inheritance. Instead it is a lattice of classes.

The ATTRIBUTE and MAP classes described in the preceding section are special kinds of classes. They were treated first because of the important role that *attributes* and *maps* play in the user definition of classes. This section shows how an individual user can create new classes. The most important construct is the <association_clause> which declares that specific sets of attributes and/or maps will be valid over elements of the class. The <restriction_clause> can be used to restrict membership in this class only to entities of the <super_class> which satisfy certain constraints.

5.2. Syntax

The syntax of class declaration is subdivided into to portions. The first describes the general mechanisms for describing new classes; the second examines in detail how predicate restrictions are formed.

5.2.1. Class Syntax

<class_decl_stmt> ::= [**var**] <dict_class_entry> **isa** <super_class>
[<class_decl_body>]

<elem_instance_stmt> ::= [**var**] <dict_inst_entry> **denotes_a** <class_name> [**AND** <class_name>]*

<super_class> ::= CLASS |
<class_name> [**AND** <class_name>]*

<class_decl_body> ::= FORWARD |
[<association_clause>]*
[<restriction_clause>]
[<scope_clause>]

<association_clause> ::= **having** [<synonym> =] <association_set>

<synonym> ::= <actual_name>

<association_set> ::= <set_desig> |
<clustered_attr_enum>

<clustered_attr_enum> ::= '{' <attr_cluster> [<attr_cluster]* '}'

<attr_cluster> ::= (<attr_desig>) [, <attr_desig>]*

<restriction_clause> ::= **provided #** <predicate> **#** |
provided <boolean_method>

<remove_element_stmt> ::= **remove** <element_name>

5.2.2. Predicate Syntax

The syntax for forming <predicate>s we treat in this separate section. Basically a <predicate> is an expression in the first order predicate logic which will evaluate to either *true* or *false*. However, the rules are somewhat different to ensure that all such expressions are "safe", that they can be deterministically evaluated.

<predicate> ::=	<disjunct> [**or** <disjunct>]*
<disjunct> ::=	<conjunct> [**and** <conjunct>]*
<conjunct> ::=	<term> | (<predicate>) | <quantifier> '[' <predicate> ']'
<term> ::=	<equality_comparison> | <order_comparison>
<equality_comparison> ::=	<element> <equality_test> <element> | <data_value> <equality_test> <data_value>
<order_comparison> ::=	<data_value> <order_test> <data_value>
<element> ::=	<logical_var> | <element>.<map_desig>
<data_value> ::=	<literal_value> | <element>.<attr_desig>
<equality_test> ::=	= | !=
<order_test> ::=	< | <= | > | >=
<logical_var> ::=	<bound_var> | <free_var>
<quantifier> ::=	(**all** <bound_var> **in** <set_desig>) (**exists** <bound_var> **in** <set_desig>)
<free_var> ::=	$X | $x

5.3. Semantics

5.3.1. Class Semantics

(1) The most common superclass is simply CLASS. The next most common is a single <super_class>, in which case the class being declared *inherits* all of the associations and restrictions of its super class.

If multiple inheritance is specified with the AND option, then the declared class inherits all of the associations and restrictions of each of its super classes.

(2) If the **var** option is missing then the *literal* string constituting the <dict_class_entry> (or <dict_name_entry>) is the dictionary lookup string. If **var** precedes the declaration, then <dict_class_entry> (or <dict_name_entry>) is presumed to be a host language variable of type "string" whose current value is the corresponding dictionary name. See 5.6 for a discussion of the handling of literal and variable identifiers.

(3) The FORWARD option for a <class_decl_body> is similar to that of Pascal, and for the same reason. In order to define a *map* one must first identify the class which is its image space. If the map is a function from a class back into itself, such as the "subpart_of" relationship, this becomes difficult. The FORWARD construct conveys sufficient information to create the basic dictionary entry. Subsequently, a complete declaration must be provided.

(4) An <association_clause> associates an existing *instance* set of attributes or maps with the elements of the class. This set may be either named (presuming a previous instance declaration) or enumerated (implying creation of the instance at compile/run-time?).

There may be repeated <association_clause>s. This is necessary to associate both attributes and maps with a class. It also provides for the possibility of associating several different sets of attributes (or maps) with a class, thereby supporting a *view* concept.

It is also possible to provide an optional <synonym> for the association set. This <synonym> may be used to access individual elements of the set. The conventional synonyms *attrs* and *maps* are considered public. Any association sets so identified will be displayed on a request to describe the class. Association sets with other (or no) synonyms are treated as private.

(5) The clustered attribute enumeration permits a parenthesized enumeration of attributes, such as *{ (a, b, c) (d, e) (f) }*. This "clustering" may, or may not, be used to optimize the retrieval of attribute values.

(6) The <predicate> or user supplied <boolean_method> or a restriction clause is evaluated whenever an instance of the class is created. If it evaluates *false*, then the ADAMS statement fails.

At most one free variable is permitted in a predicate used for class declaration, and it is denoted by $x or $X. This free variable always denotes the *current* instance of the class which is being tested for class membership. It is completely analogous to the "SELF" construct which is used in several object oriented languages.

(7) Declaration of an entity instance (element) via a "denotes_a" statement, or by any other ADAMS operation, will allocate the "next" unique id to identify the instance. It will also create the "instance body" which is a record consisting of at least

a. CLASS pointer,
b. set reference counter (set membership count)
c. removal bit
d. given name (if any)

This little stub representation is required to implement the *class_of* and *name_of* system procedures (11.2), and the issue of element deletion (6.6).

(8) In as system that supports the representation of persistent data, the deletion of information can be much more difficult than its creation. In effect, the <remove_element> statement is the inverse of the <elem_instance> (or "denotes_a") statement; and the <erase_class> statement is the inverse of the <class_decl> (or "isa") statement.

But care must be taken! Readily, a class can not be erased if there exist any instances of that class. Similarly, an element can not be removed if it exists in an existent set. These are two important examples of internal database consistency that must be maintained. The use of a set reference counter in every instance body can be used to protect against the latter. In addition, a *removal bit* must be included in the representation to support deferred removal.

(See 6.4.) A reference counter which keeps track of all instances belonging to a class, and another recording all sub-class references, can also be exploited.

When a persistent element instance is created, its reference counter is set to one. The "remove" statement first decrements the reference counter; if it is then zero, the element is actually removed and its storage returned to the system.

What can not be assured, given the environment in which ADAMS exists, is that when a CLASS or an instance is deleted there will be no extant process that refers to it. This latter is a form of external consistency.

5.3.2. Predicate Semantics

(1) Atomic truth values are obtained only from equality or order comparisons. Elements can only be tested for equality; either they have the same unique id or they do not. Codomain values (e.g. <data_value>s) can also be tested for equality. In this syntactic formulation we have also allowed for order comparison, but whether this can be actually implemented is open to question.

(2) Quantification is always over existing set instances, never over an abstract class.

(3) Any named construct used in a class declaration, whether <super_class> or <set_desig> must have a scope equal to, or higher than, the current declaration. This dependence must be recorded with the named construct in its <reference_counter> so that it can not be inadvertently deleted, thereby making the declaration invalid.

5.4. Examples

The tuples and relations of the relational database illustrated in Figure 1 (section 1.5.1) could be declared as follows.

```
<<    FACULTY_TUPLE  isa  CLASS
            having attrs = { name, soc_sec_nbr, b_date, rank, dept }    >>
<<    FACULTY_REL  isa  SET of FACULTY_TUPLE  elements                  >>
<<    faculty  denotes_a  FACULTY_REL                                   >>

<<    STUDENT_TUPLE  isa  CLASS
            having attrs = { name, soc_sec_nbr, b_date, major, advisor }>>
<<    STUDENT_REL  isa  SET of STUDENT_TUPLE  elements                  >>
<<    students  denotes_a  STUDENT_REL                                  >>
```

The codomain of the *advisor* attribute is presumably the same as that of *name* so that student tuples can be joined with faculty tuples to obtain the advisor relationship. There are no maps in the relational model.

A much cleaner way of declaring relational schema, tuples, and relations is developed in Section 8 where parameterized class declaration is explored.

The following ADAMS statements use inheritance to create the FACULTY_REC class from a PERSON_REC class.

```
<<    PERSON_REC isa CLASS
            having data_fields = { name, soc_sec_nbr, b_date },
            scope is USER              >>
<<    FACULTY_REC isa PERSON_REC
            having fac_data_fields = { rank, dept },
            scope is USER              >>
```

Once the FACULTY_REC class has been declared, the *advisor* map can be declared, and it becomes possible to declare a STUDENT_REC entity which also inherits the basic properties of a PERSON_REC.

```
<<     FACULTY_MAP isa MAP with image FACULTY_REC,  scope is USER  >>

<<     advisor denotes_a FACULTY_MAP,     scope is USER      >>
<<     instructor denotes_a FACULTY_MAP,  scope is USER      >>

<<     STUDENT_REC isa PERSON_REC
                  having stu_data_fields = { major },
                  having maps = { advisor },
                  scope is USER                              >>
```

Notice that this latter declaration has two <association_clause>s, one for attributes and one for maps.

If faculty (or staff) members are also allowed to take courses, so that they are students as well, we might want to create the class

```
<<     PART_TIME_REC isa FACULTY_REC  AND  STUDENT_REC       >>
```

Entity instances in this class would inherit the attributes and maps of both super classes.

If a provision of being a "student" is that the individual have a declared major, we could add a <restriction_clause> as follows

```
<<     STUDENT_REC isa PERSON_REC
                  having stu_data_fields = { major },
                  having maps = { advisor },
                  provided # $x.major != udf(dept) #
                  scope is USER                              >>
```

5.5. Discussion

The syntax for the <elem_instance_stat> permits the designation of an element that inherits the properties of two classes, even though the corresponding "intersection class" has not been explicitly created by means of a <class_decl_stat>. This follows the discussion in [Pfa88]. Permitting statements such as

```
<<     x denotes_a DOCTOR AND PATIENT    >>
```

would undoubtedly be a convenient shorthand. But there are potential problems. Two implementation schemes are possible. One is to create an "unnamed" intersection class from the super-classes DOCTOR and PATIENT, to which *x* will not belong. The other is to support multiple pointers out of the dictionary entry for *x* to all of its class memberships. The former seems much preferable, but correctly implemented it requires a search of the dictionary to discover whether any class which multiply inherits from DOCTOR and PATIENT already exists in either a named or unnamed form. This will eventually lead to the nasty problem of synonym detection and resolution.

It might be wise to leave this feature unimplemented for a while.

Implementing the <predicate> construct in full generality at this time would seem to be quite difficult. However, there does not appear to be any real syntactic or semantic limitations.

These examples graphically demonstrate how useful inheritance can be in simplifying the definition of classes.

The handling of literal and variable identifiers in ADAMS is quite different from traditional programming languages where it is customary to "quote" literal strings. For example, in the statement

```
<<     advisor denotes_a FACULTY_MAP,    scope is USER>>
```

both *advisor* and *FACULTY_MAP* are literal strings. This can be quite confusing at first. In [Klu88] we suggested changing the syntax to read

```
<<     "advisor" denotes_a "FACULTY_MAP",       scope is USER>>
```

but this suggestion seems ill-advised. It would make the declaration of ADAMS names much clearer, but it would make their subsequent use more awkward. In particular, every map and attribute reference would have to be quoted, as in

```
<<     fetch into fac_name from x."advisor"."name"     >>
```

Observe that in most literal strings in traditional programming languages are not quoted. Numeric literals are not quoted because they can be recognized by their form. Literal function and procedure names are not quoted because they are declared, or are otherwise recognizable from the context. The ADAMS policy has been to assume that every *non-reserved* string in an ADAMS statement is a literal; that is, it is the literal name of an ADAMS element, unless the string is explicitly declared to be a variable. The two ways that this is done are

(1) by using the **ADAMS_var** statement to declare the identifier to be a host language variable of type UNIQUEID; and

(2) prefixing a host language string variable with **var** in *isa* or *denotes_a* declaration statements.

6. Sets

6.1. General Description

Sets are the fundamental ADAMS structure. Indeed, in keeping with our goal of simplicity, they are the only aggregation structure. Still there are significant semantic problems associated with their implementation. These arise primarily from (1) set operations over entities of different classes in the class hierarchy, and (2) entity deletion.

Sets are fundamental. But sets are not an easy concept to emulate.

6.2. Syntax

The Syntax of this section is broken into two sections, that of set denotation followed by that of set manipulation statements.

6.2.1. Set Denotation

<set_decl_stmt> ::= <set_class_entry> **isa SET**
of <class_name> **elements**
[<association_clause>]*
[<restriction_clause>]
[<scope_clause>]

<set_class_entry> ::= <dict_class_entry>

<set_instance_stmt> ::= <set_entry> **denotes_a** <set_class>
[<initial_clause>]

<set_class> ::= <class_name>

<initial_clause> ::= **consisting of** <set_desig>

<set_desig> ::= <set_name> |
<enumerated_set> |
<association_set> |
<retrieval_set> |
'NULLSET'

<association_set> ::= <element_name>-><set_name> |
<element_name>-><synonym>

6.2.2. Set Manipulation

<looping_statement> ::= **for_each** <variable_name> **in** <set_desig> **do**
[<host_language_statement>]*
[<ADAMS_statement>]*

<end_loop_statement> ::= **exit_loop**

<set_assign_statement> ::= **assign_to** <set_name> **from** <set_desig>

<set_copy_statement> ::= **copy_to** <set_name> **from** <set_desig>

<make_empty_stmt> ::= **make_empty** <set_name>

<insert_statement> ::= **insert** <element_name> **into** <set_desig>

<delete_statement> ::= delete <element_name> from <set_desig>

<union_statement> ::= <set_name> is_union_of <set_desig> [, <set_desig>]*

<intersect_statement> ::= <set_name> is_intersection_of <set_desig> [, <set_desig>]*

<complement_statement> ::= <set_name> is_complement_of <set_desig$_1$> wrt <set_desig$_2$>

6.3. Semantics

6.3.1. Set Denotation

(1) Only in the set instantiating statement is a initialization clause <initial_clause> permitted, which will initialize the newly denoted set to some existent set. The latter may be a named set, or it may be an enumerated set that is completely designated in the instantiating statement, or it may be a created set in the form of a <retrieval_set>.

(2) NULLSET is the literal name of the empty set. Like all literals, it must be quoted.

6.3.2. Set Manipulation

(1) A set is implemented by a structure (possibly an O-tree) which denotes what elements (e.g. which unique id's) constitute the set. It is a set of *references* to its constituent elements.

It is anticipated that the constituent elements of most sets will exist on distinct storage devices.

(2) To reference an association set, either the name of the set must be explicitly known, or a synonym, which was established in the class declaration, must be used.

(3) A set loop statement is a true iteration statement, it performs the enclosed set of statements for each element in <set_desig>. Behavior will be unpredictable if the composition of <set_desig> is altered in the course of the loop.

The initial **for_each** initializes a looping statement which sets the <variable_name> equal to each element in <set_name> in turn and then executes any following host language and/or ADAMS statements up to the closing <e_delim> ">>".

(4) The loop variable, <variable_name>, need not be declared, since its class is completely specified by the class of elements in the existing <set_desig>.

(5) The **exit_loop** statement is exactly analogous to a "break" statement in C. It permits the immediate exit from the innermost set loop.

(6) The class of the destination <set_name> of a set assignment, or copy, statement must be the same as, or higher in the hierarchy, than the class of the source <set_desig>. Thus, if *people* is a set of PERSON entities, then either

```
<<    assign_to  people from undergrad        >>
```

or

```
<<    copy_to  people from tenured            >>
```

will succeed but

```
<<    copy_to  undergrads from people  >>
```

and

```
<<    assign_to  untenured from undergrad     >>
```

will fail. The last statement, in which the classes of *untenured* and *undergrad* are not

comparable can not make semantic sense, since the elements in *untenured* would not have several FACULTY attributes defined over them, while having several STUDENT attributes defined. It would violate the class system.

The preceding "copy_to" statement, in which the destination <set_name> is lower in the hierarchy than the source <set_desig>, could be semantically interpreted to mean: "for each element of class PERSON in the set *people*, create a corresponding element of class STUDENT in the set *undergrads*. Duplicate all of the PERSON attributes from the source element, and set all remaining STUDENT attributes to 'undefined'." However, no such interpretation would make sense for a set assignment with the same two operands; so we prefer apply the rule above to both statements.

(7) A set with persistent scope can not have members whose scope is LOCAL. Else persistent references would disappear when the creating process terminates.

Clearly, any set can have elements whose scope is higher that the scope of the set. For example, a local set can reference persistent elements. Less obvious is whether a set should be allowed to reference persistent elements of lower scope. Such a mechanism could be viewed as compromising the security of USER elements. Or it could be viewed as a mechanism for exporting USER elements. Our implementation will assume the latter, and allow a set with persistent scope to include any elements of persistent scope.

(8) Insertion of an element of the set must

a) check that the element is of a class that can belong to the set,
b) check that the element has LOCAL scope if the set has LOCAL scope, and
c) increment the set reference counter of the element, if the set is persistent.

We employ the latter rule, so that on process termination, ADAMS will not have to decrement the set reference counter of all elements that were included in temporary LOCAL sets. But it has a consequence discussed in section 6.5.

(9) Set assignment is a copy by reference. That is, the set of references constituting the source <set_desig> replaces the set of references that had constituted the destination <set_name>.

All elements of the destination set must be first "deleted", that is their set reference counters decremented, then replaced with pointers to the elements of the source set, each of whose reference counters are incremented. Note that reference counters of elements in LOCAL sets will not be altered.

(10) Set copy is a "shallow" copy. That is, for each element denoted by the source <set_desig>, an exact copy with a new unique id is created and "inserted" into the destination <set_name>. Any existing references in the destination set are lost.

If the source <set_desig> is the NULLSET, then this behaves as if it were a set assignment.

(11) Deletion of an element from a set does not in general remove the element from the system. It does, however, decrement the set reference counter of the element. If as a result the reference counter is zero, and if the removal bit has been set, then the element is physically removed. *if the set has persistent scope.*

(12) When a set instance is declared (with a "belongs_to" statement), it is automatically empty. The <make_empty> statement will delete any elements from an existing set. Note that the following three ADAMS sequences

```
<<     make_empty  S        >>


<<     assign_to  S  from NULLSET >>


<<     for_each  x  in  S  do
<<           delete  x  from S    >>
       >>
```

are all equivalent.

(13) Like assign and copy, the set operators *union, intersection,* and *complement* must establish the class of the result within the class hierarchy. The result of a relative complement will belong to the *same* class as the class of <set_desig$_1$>. The result of a union must belong to a class *above* in the class hierarchy, or the *same,* as the class of every argument <set_desig>. The result of a intersection must belong to a class *below* in the class hierarchy, or the *same,* as the class of every argument <set_desig>.

6.4. Examples

The example below is a horrible way of retrieving all undergraduate students who are majoring in CS. A <retrieval_set>, as described in the next section, would be much more efficient.

```
       char   data_value[20];
              .
              .
<<     cs_majors denotes_a STUDENT_SET          >>
<<     for_each x in undergrad do
<<           fetch into data_value from x.major       >>
             if (strcmp (data_value, "CS") == 0)
<<                 insert x into cs_majors            >>
             >>
```

The following C code implements a rather inefficient set intersection operator. The system intersection operator employed by the <intersect_statement> is much better; we present this only to illustrate principles of set manipulation and ADAMS coding.

```
intersect  (Z, X, Y)
<< ADAMS_var Z, X, Y       >>
       /*
       **  This procedures forms a set Z which denotes those elements
       **  belonging to both the sets X and Y (i.e. their intersection).
       */
       {
<<     ADAMS_var  z         >>

<<     assign_to  Z  from X       >>
<<     for_each  z  in Z  do
             if(!member_of(z, Y))
<<                 delete  z  from  Z  >>
             >>
       }
```

The O(n) algorithm is trivial; let Z initially be all of X and strike out those elements which are not also in Y. The *member_of* function is described in section 11.

The treatment of element removal can be illustrated by the following example. Note that these statements need not occur in the same process!

```
<<     x  belongs_to  Q     >>
         .
         .
         .
<<     insert  x  into  S   >>
         .
         .
         .
<<     remove  x            >>
         .
         .
         .
<<     delete  x  from  S   >>
```

If x and S have persistent scopes, then on completion of the second statement the reference counter of the element x will be 1 (because of the insertion). Consequently, the following request to remove x as an element will be deferred, only its removal bit will be set. When, subsequently the element is deleted from S, its reference counter will have been decremented to zero and because its removal bit has been set, it will be actually removed.

6.5. Discussion

The implementation of sets is going to be dicey, as some of the following comments indicate.

It is not clear how to represent a set in a distributed memory environment. In a uniprocessor, or a shared memory, environment a set could be represented by a single element referencing structure. In a multi-memory, multi-device environment should the defining element membership structure of the set also be distributed?

Set assignment could be denoted by a more traditional assignment operator symbol, such as :=. Then we could have

```
<dest_set> := <source_set>
```

instead of

```
assign_to <dest_set> from <source_set>
```

But is this wise? Does the different syntax serve to focus the user's attention on the nature of the assignment, or is it just distracting?

This is the place to explore the implementation of a relational *project* operator, $\Pi_X(set)$. The problem really has to do with the class hierarchy. Elements in the set $\Pi_X(set)$ belong to class X, where X belongs somewhere between the class of "set elements" and the universal class CLASS. But how is such a class created and inserted into the hierarchy?

In the element removal example of the preceding section, the element x was not actually removed by the <remove_element> statement, because its set reference counter was non-zero. But if S was a LOCAL set that counter would not be incremented. The element x would be removed even though a reference to it still occurred. This is a clear anomaly. But, if the set S is LOCAL, the insert, remove, and delete statements must all occur in the same program—so it is a clear *programmer error*, not an ADAMS error!

7. Attribute and Map Inverses

7.1. General Description

ADAMS attributes and maps are, by design, single valued. Expressions of the form *<element_desig>.<attr_desig>* and <element_desig>.<map_desig> denote a single data value or ADAMS element, respectively. But the essence of much database processing is the access to those elements, or entities, which have some specified attribute (or map) value. For example, we might want to access all STUDENT entities whose *major* is 'CS'. We want to denote the *inverse image* of the data value 'CS' under the *major* attribute function. In general, the inverse image of any function is a set.

This section describes the syntax of such set denotation, which we will generally call a <retrieval_set>. This special form of set denotation could have logically been included in the preceding section, but there is sufficient material to treat it separately.

Regarding all attributes and maps as sets of triples of the form

(<element_id>, <attribute_id>, <data_value>)

or

(<element_id>, <map_id>, <element_id>)

specification of the first two components in each case will yield the unique (because both are functions) third component. An inverse operation occurs whenever the last two triple components are specified, as in

(*x*, major, 'CS')

or

(*x*, advisor, *y*)

where *y* is a unique faculty id. In both cases we want the set of all elements *x* for which the triple exists in the ADAMS database. The first case would yield all elements "who major in CS" as above.

One of the earliest treatments of data representation by means of ordered triples is the seminal LEAP system [FeR69] which simulated associative memory by hash coding. However, this triple notation by itself is syntactically incomplete. The elements { *x* } of the inverse set must all belong to some class; and that class must be specified. To see that this is really a problems consider an inverse of the form

(*x*, name, 'Chip')

The inverse element, *x*, might denote a person, a dog, or even an electronic component whose "name" is 'chip'. To be well formed, the class of the inverse elements must be specified. To be safe, the inverse elements must be restricted to a finite set.

Inverse operations are specified using a predicate syntax, not a triple syntax.

7.2. Syntax

<retrieval_set> ::= '{' <bound_var> **in** <set_desig> '|' <predicate> '}'

7.3. Semantics

(1) A retrieval set can only consist of elements. It is impossible to retrieve a set of "data values".

(2) The class of a retrieval set is well defined; it must be the same as <set_desig>.

(3) Because all elements satisfying the predicate expression are restricted to membership in <set_desig>, this retrieval expression must be "safe" (p.247, [Mai83]).

7.4. Examples

The following straight forward example retrieves CS majors. It is equivalent to

$$\{ x \} = \text{'CS'}.\text{major}^{-1} \mid_{\text{undergrad}}$$

that is, the inverse image of the *major* attribute restricted to the set *undergrad.*

```
<<    cs_majors denotes_a STUDENT_SET          >>
<<    assign { x in undergrad | x.major = 'CS' } to cs_majors      >>
```

The following is an interesting array inverse. It finds all zero elements of the array *x.*

```
<<    zeros denotes a REAL_ATTRIBUTE_SET       >>
<<    assign { f in x->attr | x.f = '0' } to zeros   >>
```

Note that *zeros* is a set of attributes. Assuming that we might like the identity of the zeros, we might expect to display their locations by

```
<<    for_each f in zeros do
            printf ("%s0, name_of(f) );
            >>
```

7.5. Discussion

The issue of order comparisons, or inequalities, in predicate expressions is still very much in the air. Suppose, in the matrix example that we wanted the identity of all negative entries, as in:

```
<<    negative denotes a REAL_ATTRIBUTE_SET    >>
<<    assign { f in x->attr | x.f < '0' } to negative        >>
```

What does the '<' mean?, less than lexicographically? or less than numerically? The latter would require either creating the attribute index using numeric keys or fetching *x.f*, converting it to a numeric value, and performing the comparison in the host language.

8. ADAMS Names and Designators

8.1. General Description

A *designator* is a symbolic string which serves to designate a single ADAMS element; it may be a data value, an attribute, a map, an entity, or a set of entities. The most basic designator is a *name*. By an ADAMS name we mean a literal string that identifies an ADAMS element. In all host languages the literal sequence, -2.53, denotes the unique real value '-2.53', or more correctly the binary string whose conventional interpretation is that real value. In ADAMS, literals are names, each of which denotes a distinct entity, that are entered into the dictionary for subsequent use.

But simple "literal names" turn out to be inadequate for denoting and describing vast collections of persistent data. We find we want to be able to parameterize names and to be able to subscript them as well. Moreover, as noted by [KhC86] naming is not the only way of identifying objects. Objects, or entities, may be designated in a variety of ways. A variable may be used to designate different entities, depending on its current value. (In ADAMS, variables function effectively as pointers.) An entity may be designated by an expression, which is evaluated at run-time. A set entity may be designated by retrieval expression which both creates the set and denotes it as well.

This section details the various ways that ADAMS designators may be constructed. Since the designation, or identification, of data and sets of data, is central to ADAMS role in storing and accessing of large databases, this syntax is crucial. And since naming is a key form of designation, a flexible syntax for forming names is important.

8.2. Syntax

<char_seg> ::=	<string of letters and/or digits>
<param_seg> ::=	$<ordinal_number>
<pattern_seg> ::=	<char_seg>| <param_seg>
<dict_class_entry> ::=	<pattern_seg> [_<pattern_seg>]*
<actual_name> ::=	<char_seg> [_<char_seg>]*
<subscript_decl> ::=	<subscript_pool_name> [, <subscript_pool_name>]*
<dict_instance_entry> ::=	<actual_name> | <actual_name> '[' <subscript_decl> ']'
<subscript> ::=	<subscript_value> [,<subscript_value>]*
<subscripted_name> ::=	<actual_name> '[' <subscript> ']'
<class_name> ::=	<actual_name>
<element_name> ::=	<actual_name> | <subscripted_name>
<ADAMS_var_name> ::=	<actual_name>
<variable_list> ::=	<ADAMS_var_name> [, <variable_list>]
<variable_decl_stmt> ::=	**ADAMS_var** <variable_list>

<element_desig> ::=	<element_name> | <variable_desig> | <element_desig>.<map_desig>
<attr_desig> ::=	<element_desig>
<map_desig> ::=	<element_desig>
<set_desig> ::=	<element_desig> | <inverse_set> | <enumerated_set>
<range> ::=	<subscript_value> | <subscript_value> .. <subscript_value>
<range_subscript> ::=	<range> [, <range>]*
<enumeration_elem> ::=	<element_name> | <actual_name>'['<range_subscript>']'
<enumerated_set> ::=	'{' [<enumeration_elem> [, <enumeration_elem>]*]* '}'
<var_assign_stmt> ::=	<ADAMS_var_name> **denotes** <element_desig>

8.3. Semantics

(1) ADAMS names are composed of segments separated by underscore. The segment may consist of characters (letters and/or digits) or it may be a formal parameter of the form $n.

"Actual" names have no parameter segments. Similarly, "instance" names, which are used to actually denote entities in ADAMS storage, may have not parameter segments but may be subscripted. Codomain, subscript pool, and variable names may be neither parameterized nor subscripted.

(2) A dictionary "class_name" is a pattern asserting that all names with this pattern have the declared properties of the class. Such dictionary names with parameter segments can be used only in class definition statements, such as

<char_seg>_$1_<char_seg> isa ...

or

$1_<char_seg>_<char_seg>$2 isa ...

The parameter segment, $n, can match any character segment, and that character segment (actual parameter) will replace the parameter segment (formal parameter) throughout the remainder of the definition, wherever it appears again. Note that a single (formal) parameter segment can never be replaced by a segmented (actual) string.

These dictionary "class names" provide a mechanism for parameterized name formation. Only the pattern need be stored in the dictionary. Instantiation names can not be parameterized.

Since, by itself a parameter segments such as $1 would match all (unsegmented names), a <dict_class_name> must contain at least one character segment.

(3) All ADAMS variables in a program segment must be declared, otherwise the character string is assumed to be a instance name that exists in the dictionary.

(4) To instantiate an entity using a "denotes_a" statement, one need only establish a one-to-one correspondence between the denoting name, which may be subscripted, and a unique element id. There is no need to actually allocate storage for the entity. If the dictionary instance name is a simple <actual_name>, then a unique id is allocated for that name. If the instance name is subscripted, e.g. *x*[*<subscript_pool>*, *<subscript_pool>*], then as before a unique id is associated with the <actual_name>. This can be modified by a distinct integer suffix for each of its *possible* n subscript values. Thus the correspondence is defined implicitly, rather than explicitly.

Dictionary lookup of instance names, even if subscripted, is always by the initial <actual_name> portion.

(5) In ADAMS, even if a name is subscripted with values form several subscript pools, it is the n-tuple of all values that is treated as the "subscript".

(6) Since attributes, maps, and sets are all ADAMS elements (or entities), their designators all have the form of a general <element_desig>. However, there are situations, such as <value_desig>, where one must use a <attr_desig> as one of its components. Such constraints are not easily captured in the BNF syntax we are using.

(7) It is assumed that the compiler has access to the dictionary. It must, in order to verify instance names. Consequently, all instance names can be replaced with the corresponding unique id's at compile time.

(8) It is also assumed that compilation creates the LOCAL version of the dictionary in the form of a loadable program unit. It has all the needed information. Consequently, sophisticated pattern matching will have no run-time penalty.
Where new names are declared with permanent scope (USER, GROUP, or SYSTEM) these are marked, and actually copied into those portions of the dictionary on successful completion of the program.

(9) An <enumerated_set> is just that, the enumeration of the literal names of zero, or more, constituent elements. For convenience, we also allow the use of a <range> of subscript values in this construct as a simple way of declaring enumerated sets. This is the only use of the <range> construct.

Both <subscript_value>s of the <range> must exist, and the first must precede the second in the subscript pool.

(10) A <ADAMS_var_name> denotes a element (more particularly, its unique id). It is unnecessary to declare the class of a <ADAMS_var_name> because it can be determined by the context (as in a set loop construct). Actually <ADAMS_var_name>s will be typed in the host language, as in the C declaration

```
UNIQUEID      <ADAMS_var_name>;
```

Thus variable names, and the variable assignment statement can be used to provide an interface between ADAMS designations and host language procedures as in

```
<<      <ADAMS_var_name> denotes <set_desig>          >>
        CALL SORT (<ADAMS_var_name>)
```

The host language type, UNIQUEID, of <ADAMS_var_name>s may be environment dependent.

8.4. Examples

The first three statements illustrate how ADAMS declares generic relations and relational tuples. The last two statements then use these SYSTEM declarations to define an instance

relation, *faculty*, as illustrated in section 1.5.1. This relation is initially empty.

```
<<    SCHEMA isa SET
            of ATTRIBUTE elements, scope is SYSTEM          >>

<<    $1_TUPLE  isa CLASS
            having  attributes = $1 , scope is SYSTEM
            provided #$1.class_of = 'SCHEMA'#               >>

<<    $1_RELATION isa SET
            of $1_TUPLE elements,  scope is SYSTEM          >>

<<    FACULTY denotes_a SCHEMA
            consisting of { name, soc_sec_nbr, b_date, rank, dept },
            scope is USER                                   >>

<<    faculty denotes_a FACULTY_RELATION, scope is USER     >>
```

In the example of Section 4.4, a map class with the class name FACULTY_MAP was declared so that instance maps called *advisor* and *instructor* of this class could be established. A parameterized class declaration, such as below, would have been preferable.

```
<<    $1_MAP isa MAP with image $1_REC,  scope is USER      >>

<<    advisor denotes_a FACULTY_MAP,     scope is USER      >>
<<    instructor denotes_a FACULTY_MAP,  scope is USER      >>
```

While this offers no economy in the definition of these two specific maps, it does provide a mechanism for defining the *student* and *course* maps without having to additionally declare STUDENT_MAP and COURSE_MAP. The following instantiations would sufficient.

```
<<    student denotes_a STUDENT_MAP, scope is USER   >>
<<    course  denotes_a COURSE_MAP,  scope is USER   >>
```

In the following example we will use subscripting to declare (a) the class of all doubly subscripted real arrays, or matrices, and (b) a particular 5x8 matrix denoted by *x*.

```
<<    $1_ATTRIBUTE isa ATTRIBUTE
            with image $1,
            scope is SYSTEM             >>

<<    val[Zahlen, Zahlen] denotes_a REAL_ATTRIBUTE,
            scope is USER               >>

<<    REAL_$1_X_$2_MATRIX isa CLASS
            having  attr = { val[1..$1, 1..$2] },
            scope is USER               >>

<<    x denotes_a REAL_5_X_8_MATRIX
            scope is USER               >>
```

Subsequently, procedures can make use of the permanent data that is denoted by elements of *x*. For example,

```
<<    fetch into a[3, 5] from x.val[3.5]      >>
```

8.5. Discussion

Literals are much more important in ADAMS than in traditional languages. In host programming languages, literal strings typically "denote themselves", whether they are numeric literals or quoted literals. In ADAMS, a literal string (or name) denotes a single identifiable object, or class. The dictionary is simply a mechanism for looking up the meaning of these literal names.

It is important to note that instance names and variables have the same form, so that it is impossible to distinguish them within the context of a single ADAMS statement. This is not true in many other programming languages. In these languages, literals are recognizable because they are 1) numeric, 2) quoted, or 3) used in a definable context (e.g. procedure names). Two important exceptions are named constants in Pascal and defined constants in C. Their literal nature is discoverable only by compilation. ADAMS employs this paradigm.

Name segments that are to function as "actual parameters" in a parameterized <dictionary_name> are not distinguished as such. This makes the resulting names more natural, but it also can lead to problems. For example, which of the two dictionary name patterns, $1_RELATION or R_$1, should R_RELATION match? There are several, somewhat unelegant, ways of resolving this (e.g. actual parameter segments can not be capitalized) but I am inclined to wait and see how the present scheme works out.

The syntax of this section has developed the differences between a <dict_class_entry>, a <dict_inst_entry>, a <class_name>, and a <element_name>. The first two represent the form of names as they are entered into the dictionary. The former can be parameterized with $n segments; the latter can specify subscript domains (or pools) that provide subscript values. The last two represent the form of names as they are used in a program to reference dictionary entries. An <element_name> can be subscripted, and <class_name> can not—it must be an <actual_name> comprised of character segments.

In the preceding sections, we have been careful to insure that the syntax conforms to these rules, but we have also used the words attr, map, and set to emphasize other aspects. The following table summarizes the various synonyms we have used in preceding sections

Defined by	Referenced by
<dict_class_entry>	<class_name>
<attr_class_entry>	<attr_class_name>
<map_class_entry>	<map_class_name>
<set_class_entry>	<set_class_name>
<dict_inst_entry>	<element_name>
<attr_entry>	<attr_name>
<map_entry>	<map_name>
<set_entry>	<set_name>

9. The Dictionary

9.1. General Description

The dictionary has just two functions. To associate with each literal ADAMS name either (a) the properties of any entity in the class, if it is a CLASS name, or (b) the unique id corresponding to that literal name.

9.2. Syntax

The dictionary concept adds only one construct to the ADAMS syntax; that is the *scope* construct. But it also adds to essential dictionary manipulation statements.

<scope_clause> ::= **scope is <scope>**

<scope> ::=	SYSTEM |
	GROUP |
	USER |
	LOCAL
<rescope_stmt> ::=	**rescope** <name> as <scope>
<erase_entry_stmt> ::=	**erase** <dict_entry>
<dict_entry> ::=	<dict_class_entry> |
	<dict_inst_entry>

9.3. Semantics

The semantics associated with the dictionary and dictionary maintenance are more fully discussed in [PFW88]. Here we only mention some of the highlights.

(1) Name scopes are hierarchical. Names declared to have SYSTEM scope are available to all users. Those declared GROUP are available to all members of the group, while USER names are private to that user. LOCAL names are not persistent; they exist only for the duration of the program.

(2) To a compiling, or executing, program the dictionary can be viewed as consisting of four sub-directories—its *local*, *user*, *group*, and *system* sub-directories. For name resolution, the local sub-directory is searched first, then the user, group, and system sub-directories, in that order. Consequently, a user can "redefine" any name declared at a higher scope.

(3) Insertion of a new <dictionary_name> into a sub-directory can succeed only if that <dictionary_name> does not already exist in that sub-directory *or* in any higher sub-directory that *is being referenced* along a path through the sub-directory. This requires keeping track of name reference by user id's.

(4) Dictionary names can not be deleted if they are currently being referenced by entries in other sub-directories.

(5) Rescoping a name can be viewed as a process of deleting and then adding it again; but not quite. It must be conducted with respect to all other users, ignoring the user issuing the command.

9.4. Discussion

All ADAMS statements which manipulate the dictionary, including

<class_decl_statement>s	**isa**
<elem_instance_statement>s	**denotes_a**
<rescope_statement>s	**rescope**
<erase_entry_statement>	**delete**

must appear in the same source file as the main program which will invoke them. This curious restriction is imposed by the a desire to optimize performance. But, before examining why we impose this restriction, lets consider its consequences. With this restriction, no **isa** or **denotes_a** statements creating either class or instance entries can appear in any separately compiled code, such as utility routines. This would seem to be a serious restriction. But consider that no **isa** or **denotes_a** statement involving literal names can, in general, be executed twice! The names are persistent. Requiring such statements to be with (or even as) the main program involves little hardship. More general, parameterized **isa** or **denotes_a** statements in which the <dict_entry> is a host language string variable would be precluded from pre-compilation, and this might be irksome. For example, one can imagine a general interactive class declaration module in which a user is prompted for various components needed to define the class.

The reason for this restriction comes from the following. At run time, the **open_ADAMS** statement, among other initialization functions, attaches the working dictionary comprised of the three persistent *user*, *group*, and *system* sub-dictionaries, together with an empty *local* sub-dictionary. In the course of execution the running ADAMS program may add entries to this local, temporary dictionary. It will save considerable run-time overhead if the compiler actually creates this local dictionary at compile time, and simply prepends it to the object code. It can then then be simply loaded by the initial **open_ADAMS** statement and the run-time equivalents of the declaration statements can be no-ops. Moreover, this permits the compiler to replace all literal names with the corresponding element UNIQUID's to eliminate most run-time dictionary lookups. If a persistent class, or entry, declaration is made the same procedure is followed, except that instead of a no-op the run-time equivalent becomes a rescope action which may, or may not, succeed at the time of execution. In order, to build such a *local* sub-dictionary at compile time, the pre-processor must see all of the relevant declarations; hence they must be in a single source code file, the same one which will issue the **open_ADAMS** command.

We have indicated that this restriction has been imposed for the sake of efficiency. We should note that it is also a necessity. The preprocessor would have to create some form of local dictionary to perform type checking on the ADAMS code it is scanning. Moreover, we could not allow reference to a non-local dictionary entry which has not yet been entered, but which *will* be entered by a separate module which *will* be run before the current code.

10. Transactions

10.1. General Description

ADAMS provides the user with nested transactions based on the well-known model of Moss [Mos85]. These transactions are designed only to provide *concurrency control.* Fault tolerance and reliability control will be buried within the ADAMS implementation and will not be accessible to the user. However, the casual user need not become involved with either the transaction concept or concurrency control at all.

A *transaction* is an ADAMS element (entity or object) belonging to the system defined class TRANSACTION. A *root transaction* with LOCAL scope is created automatically by the <open_ADAMS_statement> and automatically committed (if possible) by the <close_ADAMS_statement>. None of the intervening ADAMS statements can modify the persistent data space unless the final committment is successful. By creating nested sub-transactions the user can establish whether the intervening statements within the sub-transaction are *committable.* If a sub-transaction is not committable (i.e. the <end_trans_statement> fails) the user has the option of re-executing that sub-transaction or otherwise repairing the damage. If the sub-transaction is committable (i.e. the <end_trans_statement> succeeds), it is known that none of its intervening statements can prevent commitment of the root transaction. But the actions of its statements will actually be committed if and only if the root transaction commits.

10.2. Syntax

<transaction_statement> ::= **tr_start** <trans_desig>
([<ADAMS_stmt>] | [<host_stmt>])*
tr_end

<abort_statement> ::= **abort** <trans_desig>

<lock_statement> ::= **lock** <element_desig>

<unlock_statement> ::= **unlock** <element_desig>

10.3. Semantics

The semantics of transactions depend on the following SYSTEM declarations

```
<<    TRSTATUS isa CODOMAIN
            consisting of #who_knows_what#,
            with scope SYSTEM                          >>
<<    tr_status denotes_a to TRSTATUS_ATTRIBUTE
            with scope SYSTEM                          >>
<<    TRANSACTION isa CLASS forward                    >>
<<    TRANSACTIONS isa SET of TRANSACTION elements
            with scope SYSTEM                          >>
<<    tr_parent denotes_a TRANSACTION_MAP              >>
<<    tr_subset denotes_a TRANSACTIONS_MAP             >>
<<    TRANSACTION isa CLASS
            having attr = { tr_status, [others ?] }
            having maps = { tr_parent, tr_subset, [others ?] }
            tr_start:
                  <definition of tr_start method>
            tr_end:
                  <definition of tr_end method>
            with scope SYSTEM                          >>
```

(1) Note that the declarations of *tr_status*, *tr_parent*, and *tr_subset* above presume generic parameterized declarations of the form

```
<<    $1_ATTRIBUTE isa ATTRIBUTE
            with image $1
            scope is SYSTEM              >>
<<    $1_MAP isa MAP
            with image $1
            scope is SYSTEM              >>
```

(2) The normal sequence to create a sub-transaction would be

```
<<    tr1  denotes_a TRANSACTION >>
<<    tr_start  tr1              >>
```

The first statement creates a transaction element (entity or object). The second statement actually initializes it. We separate these two functions, so that if the sub-transaction *tr1* fails to be committable, it may be reused.

(3) The <open_ADAMS_statement> creates and initializes the root transaction. But the syntax does not provide a mechanism for returning its identity. It is a "hidden", implicit transaction that is unavailable for user manipulation.

(4) The root transaction can not commit if any of its sub-transactions are uncommittable. But note that an ABORT(ed) sub-transaction is vacuously committable.

(5) ADAMS will always use time-stamping to passively enforce serializability. The optional use of a <lock_statement> permits a user to guarantee that no time-stamp reference conflict can occur on the named entity.

If <element_desig> is a set, then the set itself and each of its constituent elements is also locked. This provides an easy mechanism for granting may locks in one fell swoop. But this is only a 1 level inclusion.

(6) A subtransaction must inherit the locks of its parent; a similar inheritance must also be implemented with respect to time-stamping.

(7) When a sub-transaction, or the root transaction, terminates entities locked in that transaction are automatically unlocked. The user initiated <unlock_statement> is strictly optional.

(8) To implement the above lock release, each transaction must have an associated <lock_set>. But this can not be an ADAMS set, because in general elements from distinct classes can be locked; it must be a system maintained "set".

10.4. Examples

The following example illustrates the process for granting a set of locks on "all the undergraduate CS majors", presumably for the purpose of a massive update.

```
<<    cs_majors  denotes_a  STUDENT_SET        >>
<<    assign  { x in undergrad | x.major = 'CS' } to cs_majors     >>
<<    lock  cs_majors                                  >>
```

10.5. Discussion

There is no provision for deadlock detection in the ADAMS syntax. Should there be?

Is the "unlock" option unwise? Moss requires his nested transactions to retain the lock until the entire transaction terminates. Moreover, suppose a set of elements, such as *cs_majors* is locked, and in the course of processing elements of the set are either inserted or deleted. How would an

```
<<    unlock  cs_majors   >>
```

statement be interpreted? Would elements that have been deleted from the set be "unlocked"? Should elements that are inserted into a set be automatically locked, and those deleted

automatically unlocked? Both seem risky. A reasonable approach might be to associate with each process an "invisible" global *lockset* consisting of all locks obtained by the process. An unlock command would remove all locks in the intersection of *lockset* and the set denoted in the unlock statement. All remaining locks in *lockset* would be automatically removed on process termination.

Moss requires that only leaf transactions modify the database? Is this a necessary characteristic of nested transactions? Can it be enforced?

11.2.1. SET Functions

is_member_of (<ADAMS_element_var>, <ADAMS_set_var>);
returns true if the <element> is a member (or element) of the specified <set>.
is_empty (<ADAMS_set_var>);
returns true if the <set> is empty, and false otherwise.
card (<ADAMS_set_var>);
returns the integer cardinality of the specified <set>.

11.3. Other Predicates

same_element (<ADAMS_elem_var>, <ADAMS_elem_var>) returns true if the two variables denote the same element.
ADAMS_success;
returns true if the last executed ADAMS statement succeeded.
ADAMS_fail;
returns true if the last executed ADAMS statement failed.
This and the preceding function simply test the ADAMS_status register.

11.4. Discussion

Should all system procedures (or methods) be clearly identifiable, say with an embedded dollar sign, etc.

12. References

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

[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.

[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.

[DGS88] D. J. DeWitt, S. Ghandeharizadeh and D. Schneider, A Performance Analysis of the Gamma Database Machine, *Proc. SIGMOD Conf.*, Chicago, June 1988, 350-360.

[FeR69] J. Feldman and P. Rovner, An Algol-based Associative Language, *Comm. of the ACM 14*, 10 (Oct. 1969), 439-449.

[HuK87] R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *Computing Surveys 19*, 3 (Sep. 1987), 201-260.

[JeW75] K. Jensen and N. Wirth, *Pascal: User Manual and Report*, Springer-Verlag, New York, 1975.

[KhC86] S. N. Khoshafian and G. P. Copeland, Object Identity, *OOPSLA '86, Conf. Proc.*, , Sep. 1986, 406-416.

11. System Procedures

The basic imbedded structure of ADAMS dictates that an ADAMS statement, denoted by its beginning and ending delimiter will be converted into corresponding host language code and/or procedure calls by the preprocessor. But in a complete interface there invariably arise occasions when a host language statement must invoke some predefined ADAMS procedure. These are typically of two forms: (1) to extract information from the dictionary for comparison, testing, or display; or (2) to test some aspect of the system. The latter will be boolean (or LOGICAL) functions.

We call these "system procedures". It would be equally true to call them "methods", especially the latter functions which are clearly associated with specific ADAMS classes.

Since a system procedure (or method) is a host language construct, all formal and actual parameters must be recognized in the type structure of the host language. The ADAMS <variable> construct is important here. It is the only ADAMS construct which must have a predefined corresponding host language type. (The correspondence between codomains and host language types is not pre-defined. It is established with fetch and store methods.)

11.1. Dictionary Interrogation

Many of the procedures below return *strings* as their functional value—that is, a "string" in the sense of the host language. Others accept strings as their argument.

class_of (<ADAMS_element_var>);
returns the class of the designated instance element, as a string. This function must be defined for all elements.

name_of (<ADAMS_element_var>);
returns the name of the designated instance element, as a string. Note that most instances will be unnamed.

unique_id_of (<ADAMS_element_var>);
returns the unique_id identifying every ADAMS element in a printable string form.

class_of_member (<ADAMS_set_var>);
returns the class of the members (elements) of the designated set.

image_of (<ADAMS_function_var>);
returns the class of image objects of the designated function, either attribute or map.

is_instance_name (<string>);
returns true if the name denoted by the <string> is the name of an instance in the user's dictionary.

is_class_name (<string>);
returns true if the name denoted by the <string> is, or could be, a class name in the dictionary.
(Note: because of parameterized class naming, it is impossible to always know if a particular actual name is being used as a name.

11.2. Class Functions

The following functions each return scalar values that are typed according to the host language's conventions, usually either *integer* or *boolean;* they are functions that are associated with a particular kind of ADAMS class (or derivative class).

Notice that in every case the arguments are ADAMS variables, that is entity identifiers which have been cast into a specific host language variable form.

[Klu88] C. Klumpp, A C Interpreter for the ADAMS Language, IPC Tech. Rep.-88-005, Institute for Parallel Computation, Univ. of Virginia, Aug. 1988.

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

[Mos85] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, MA, 1985.

[PSF87] J. L. Pfaltz, S. H. Son and J. C. French, Basic Database Concepts in the ADAMS Language Interface for Process Service, IPC Tech. Rep.-87-001, Institute for Parallel Computation, Univ. of Virginia, Nov. 1987.

[Pfa88] J. L. Pfaltz, Implementing Set Operators Over a Semantic Hierarchy, IPC Tech. Rep.-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, *4th International Hypercube Conference*, Pasadena, CA, Jan. 1988.

[PFW88] J. L. Pfaltz, J. C. French and J. L. Whitlatch, Scoping Persistent Name Spaces in ADAMS, IPC Tech. Rep.-88-003, Institute for Parallel Computation, Univ. of Virginia, June 1988.