IPC Technical Report 93-003

# The ADAMS Language: A Tutorial and Reference Manual

John L. Pfaltz

IPC-93-003
April, 1993

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

**Abstract:**

This report describes the ADAMS language as it would be used by an applications programmer. It does not describe how ADAMS is implemented.

The first three sections assume no knowledge of ADAMS whatever, and are quite tutorial in nature. Only basic, introductory concepts are covered. The remaining sections, although still tutorial, presume some familiarity, e.g. having coded simple programs of the same complexity as those in Section 3. The treatment in these sections is definitive, so that the report can be also used as a reference manual.

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

## 1. Introduction

Although the acronym, ADAMS, stands for **A**dvanced **DA**ta **M**anagement **S**ystem, ADAMS should not be viewed as a database *system*, but rather as a complete *language* based on a formal definition of persistent data [Pfa92]. It is a way of talking about persistent data that a user has inserted into the ADAMS data space. It is a way of describing the structure of that data, that is the relationships between different elements of the data space, as well as a way of accessing, in a largely non-procedural way, elements within the data space.

Because it is a language, it has syntactic rules defining what are, and what are not, well defined ADAMS statements. It also has rules regarding the formation and scope of names, those symbolic tokens that denote particular elements of data, as well as rules regarding formation of literals and other constants within the data space. Consequently, learning ADAMS is much like learning any computer language, such as Fortran, Pascal, C, or C++. Because ADAMS is also a programming language, knowledge of any of these languages will be of value in learning ADAMS, but beware, there are significant differences. Similarly, because ADAMS grew out of a database tradition, some of its terms were borrowed directly from the relational model of data [Cod70, Mai83]. A knowledge of relational database theory will provide valuable intuition, but again beware, ADAMS is not a relational database language, *per se*.

Some of the features of ADAMS are the result of deliberate design decisions, others are merely artifacts of its developmental history; consequently, a brief description of both the goals of ADAMS and its history can provide a useful background context for its structure. ADAMS has been developed with four basic database goals in mind [PfF93].

First, and foremost, its purpose has been to interface many different computing environments to a common, persistent data space. This arose from an experience many years ago in a NASA research division when we found that three groups, engineers coding in Fortran, CAD designers using Pascal, and mathematicians doing finite element analysis in assembly language, could not share data regarding their common project — the design and analysis of airfoils. Consequently, a conscious decision was made that the applications code should be that of the programmers choice, and that ADAMS would have to interface to it. This, in turn, dictated that it be an embedded system based on procedure calls; however, we wanted to present the programmer with a cleaner language consisting of embedded statements which the system itself would convert to the appropriate procedure calls.

Second, there should be different levels of data sharing and data privacy within the persistent data space. Some data might be widely shared throughout the system, some data might be common only to selected groups of users, while still other data might be completely private to a particular user.

Third, we wanted this persistent data space to be an extension of, or even a replacement of, the individual programmers file system. Except for archival purposes, scientific data needs do not lend themselves to systems managed by a database administrator. Not only does a scientist want to update existing data sets with respect to a transaction paradigm, he should be able (1) to create and name completely new persistent data sets, (2) dynamically reconfigure existing data sets to include new attributes or delete unnecessary attributes, and (3) dynamically create or destroy relationships between data sets. In short, individual programmers should be able to completely manage their own, and certain portions of the shared, persistent data space.

Finally, we envisioned a situation where the kinds of application code that now access persistent data within this data space, could equally well access a variety of metadata. For example, one might have a program that could search through the entire data space for instances of data sets which relate measurements of dissolved oxygen to stream flow.

We did not specifically design ADAMS to be a distributed, parallel database system, although we recognized from the outset that this would probably be necessary to achieve its

design goals. Moreover, current research indicates that the approach that ADAMS has taken to database implementation may immensely facilitate both distributed and parallel processing.

Historically, ADAMS as described in [PSF88] was implemented by a very rapid prototype [Klu88]. Although severely limited, it established a rudimentary proof of concept. In particular it revealed deficiencies in the language, and suggested several modifications that were included in the version of ADAMS described in [PFG89b]. This newer version was followed by a second prototype that made use of a preliminary storage manager [Jan89], and a preliminary preprocessor [Bar89] that converted source code with embedded ADAMS statements into purely host language code. It is important to note that, because of manpower, both this earlier, and the current, preprocessor totally ignore the context provided by the host language code. We simply did not want to expend the effort to parse it! But, if the preprocessor *did* parse the surrounding host language code, there are several places where the ADAMS syntax could be changed — in some cases making it simpler.

These two efforts, together with several others, culminated in the version described in [PFG89a]. Considerable experimentation and development was based on this version of ADAMS: a Fortran version was created [Wat90], an interactive version was implemented [Rie90], and a SQL interface developed [Cle91]. From the experience gained with these projects, yet a third version of the ADAMS language was created, as described in [PFG91].

The language described in this report is very close to the latter, although there have still been a few changes. Since ADAMS is an evolving language, there may still be a few more. However, it is attaining a relative stability that makes writing a tutorial introduction worthwhile. Current research and development has been largely devoted to changing the lower levels of the system to make it more efficient rather than changing the language itself, as the user sees it. But the reader is warned, there may be slight divergences between constructs described in this report and the constructs in any particular version.

We have not yet achieved all of our design goals. There are certain constructs described in Section 4 which have not yet been implemented, again because of available manpower. Such unimplemented constructs are described in this report because they are part of the language, but we indicate their incomplete status.

## 2. Informal Tutorial Presentation

Because ADAMS is a language, rather than just a database system, it has a formal syntax consisting of constructs such as <*elem_desig*>'s, <*set_expr*>'s, <*value_desig*>'s, and <*predicate*>'s. But this syntax (c.f. Section 7) is not easy to understand out of context. It is not the way to learn a language. This section consists of an informal presentation of basic terminology, together with a number of simple examples. It is not complete; we will ignore more than we will explain.

New terms, as they are introduced, will be set in **bold face**. Examples and code fragments will be set in `courier`. ADAMS expressions included in the body of the text and references to names and expressions within code examples will be set in *italics*. For the purposes of this tutorial, we will assume that the host language is ANSI C [KeR88].

### 2.1. Elements, Attributes, and Classes

The basic concept in ADAMS is that of an **element**. An element can play many different roles in ADAMS, but for now it is probably easiest to think of it as either a "tuple" in the relational model or an "object" in the object-oriented model. Let *x*, *y*, and *z* denote arbitrary elements.

Various **attribute** functions may be defined on ADAMS elements. Let *name*, *age*, and *gpa* be attributes. Then the expressions *x.name*, *y.age*, and *z.gpa* denote the current values of these attributes, respectively, on these elements. Note that in ADAMS we use the postfix notation <*element*>.<*attribute*> rather than a prefix notation, such as *name*(*x*), *age*(*y*), or *gpa*(*z*), which is much more common in mathematics and in programming languages.

A **value assignment** statement is used to assign host language values, either variable or literal, to an ADAMS element attribute; to assign ADAMS attribute values to host language variables; or to assign one ADAMS attribute value to another. The assignment operator is denoted by **`<-`**. For example,

```
<<    x.name  <-  'John Smith'      >>
<<    | p_name char* |  <-  x.name  >>
<<    y.age   <-  | years int |     >>
<<    z.gpa   <-  x.gpa             >>
```

are assignment statements. These are our first examples of complete ADAMS statements, and we observe that they are enclosed by << ... >>. Because ADAMS statements are embedded within a host language, and because we chose not to parse all source statements, there must be some mechanism for distinguishing between ADAMS statements and host language statements. We use << ... >> as **statement delimiters**.

The element names *x*, *y*, *z*, and the attribute names *name*, *age*, *gpa* are persistent ADAMS names; they are unknown to the host language. Similarly, the variable names *p_name* and *years* are those of the host language, and because we do not parse the host language, they are unknown to the ADAMS preprocessor. Consequently, to correctly interpret a statement such as

```
<<    | p_name char* |  <-  x.name  >>
```

we must delimit the **host variable** name, *p_name*, with vertical bars | ... |. Similarly, the ADAMS preprocessor cannot know what the type of *p_name* is, so that too must be included within the delimiters. This need to delimit and explicitly type host language variable names with every use is a nuisance — it could be corrected by parsing the host language statements as well as ADAMS statements. On the other hand, because ADAMS maintains two completely separate name spaces, one can use variable names in the host language which duplicate the persistent names of ADAMS.

Every ADAMS element must belong to a **class** which defines the properties of that element. One such property are the attributes defined on the elements of the class. Classes are created by the *isa* construct. For example,

```
<<    PERSON  isa  CLASS
            having attrs = { name, age }  >>
```

defines the class of PERSON elements. Individual elements belonging to this must be **instantiated** by a statement of the form

```
<<    bill  instantiates_a  PERSON  >>
```

ADAMS also supports class inheritance. Any person has the attributes of *name* and *age*, however a student also has a *gpa*, or grade point average, attribute. We could then declare that

```
<<    STUDENT  isa  PERSON
            having attrs = { gpa }  >>
```

and instantiate *mary* to be a STUDENT by

```
<<    mary  instantiates_a  STUDENT >>
```

In these examples we have been following an ADAMS convention that is quite valuable — all classes are set in uppercase, while all elements, or instances of a class, are set in lowercase.

Attributes are also ADAMS elements, so every attribute must belong to a class. The attributes *name*, *age*, and *gpa* can be defined by the statements

```
<<    name  instantiates_a  STRING_ATTR   >>
<<    age   instantiates_a  INTEGER_ATTR  >>
<<    gpa   instantiates_a  REAL_ATTR     >>
```

where the classes STRING_ATTR, INTEGER_ATTR, and REAL_ATTR are so ubiquitous that they have been system defined. It is possible to define other attribute classes, such as *IMAGE_ATTR*, but we will discuss that later in Section 4.2.4.

Neither *bill* nor *mary* as yet have any assigned attributes, but we would expect to have several value assignment statements, such as

```
<<    bill.name  <-  'William Smith'      >>
<<    bill.age   <-  '23'                 >>
<<    mary.name  <-  | p_name char* |     >>
<<    mary.age   <-  | p_age  int |       >>
<<    mary.gpa   <-  '3.0'                >>
```

## 2.2. ADAMS Names

In the examples above, we denoted two elements by the names *bill* and *mary*, and three attributes by the names *name*, *age*, and *gpa*. Further, PERSON and STUDENT are the class names. These are ADAMS **names**; they are meaningless to statements in the host language.

ADAMS maintains its own space of persistent names, much like its space of persistent data. Once a name, such as *bill* or *mary* or *age*, is used to denote an ADAMS element, that name cannot be used to denote some other element. The binding of names to the elements is persistent. But different users are likely to want to use the same name for different elements (our personal store of appropriate mnemonic names for elements of interest is rather limited).

To mitigate the general shortage of appropriate names, and at the same time to facilitate both data sharing and data privacy, ADAMS has a hierarchical name space, which currently con-

sists of four levels.[1] They are *SYSTEM*, *TASK*, *USER*, and *LOCAL*, which we also call visibility scopes. Names at **SYSTEM** scope are visible to all ADAMS users. Those names at **TASK** scope are visible to all users participating in that task. Names at **USER** scope are visible only to the particular user, and consequently are private. Two different users can declare the same ADAMS name in their USER name spaces to denote different elements or classes. Any name occurring in one of these three name space scopes is persistent; once defined by an executing program, it is defined for any other program which uses that name and name space. ADAMS names function as constants in ADAMS statements.

A name declared with **LOCAL** scope is local to the program in which it occurs. It is non-persistent. When the program terminates, the name, together with what it denotes, disappears.

ADAMS names can only be created by *isa* statements (e.g. classes) or *instantiates_a* statements (e.g elements). In either case, a specific scope may be declared, as in

```
<<    WORKER  isa  PERSON
            having attrs = { hours }
            scope is TASK                >>

<<    tom   instantiates_a  WORKER
            scope is LOCAL               >>
```

If no scope is specified the default assumption is USER, i.e. persistent.

A hierarchical name space provides many more names. So does the capability of subscripting names, which is not discussed until Section 4.1.1. But, perhaps most important, it is not necessary to explicitly name all ADAMS elements. One can use ADAMS variables to denote elements, as discussed in the following section.

## 2.3. ADAMS Variables

An ADAMS variable, or **ADAMS_var** for short, is a symbolic name that can denote any element in the persistent data space. Its function is analogous to pointer variables that denote structures in a dynamic heap, except that we do not type ADAMS_var's. ADAMS_var's are declared by the statement

```
<<    ADAMS_var  x, y, z      >>
```

If, we now execute an **element assignment** of the form

```
<<    x  <-  bill >>
```

then *x* denotes precisely the same element that the name *bill* denotes. The variable *x* can be used instead of the explicit name in subsequent expressions, such as

```
<<    x.name  <-  'William Smith'   >>
<<    | age int |  <-  x.age        >>
```

More importantly, we can also instantiate "unnamed" (in the sense that no name is entered into the name space) elements. For example,

```
<<    x  instantiates_a  PERSON     >>
<<    x.name  <-  | p_name char* |  >>
<<    x.age   <-  | p_age  int   |  >>
```

is a way of instantiating a new element, currently denoted by *x*, and assigning attribute values to it. It will, by default, be a persistent element in the data space; but since it has no persistent name, it will be accessible by other programs only if it can be referenced by some mechanism other than its name. A common way of handling such unnamed elements is to insert them into

[1] It is likely that more levels will be implemented in the next version of ADAMS.

sets, as described in the next section.

## 2.4. Sets

Sets are fundamental in relational database systems. It is the relation, or set of tuples, on which the relational model is based. Sets are also fundamental to ADAMS; they are the means by which we aggregate data and a primary mechanism by which we retrieve data. The entire "query language" aspect of ADAMS is essentially captured by a single set construct, which we will later call a "retrieval set".

A **set** is an ADAMS element. Like every other element, it must belong to a class and it must be instantiated. A set class is declared by an *isa* statement of the form

```
<<    PEOPLE  isa  SET
            of PERSON elements      >>
```

Because elements of any class may have associated attributes, set classes may too. We might declare the set of students by

```
<<    STUDENTS  isa  SET
            of STUDENT elements
            having attrs = { count, avg_gpa }
            scope is TASK                       >>
```

where *count* and *avg_gpa* are attributes of the set as a whole, not individual elements within the set. Now we might execute

```
<<    undergrad  instantiates_a  STUDENTS >>
<<    graduate   instantiates_a  STUDENTS >>
```

to create two distinct student sets.

Elements may be inserted and removed from set elements. Earlier, *mary* was instantiated as a STUDENT element, so

```
<<    remove  mary  from  undergrad >>
<<    insert  mary  into  graduate  >>
```

would be appropriate code to change *mary*'s status from undergraduate to graduate. However, only elements which are **conformable** to the declared elements of the set may be entered. An element is conformable if it is either the same class as, or a subclass of, the class of set elements. For example, *bill* had been instantiated as a PERSON, so the statement

```
<<    insert  bill  into  graduate  >>
```

would be treated as an error by ADAMS. Of course, one could insert *mary* into a set of PERSON elements, because every STUDENT element is a PERSON element. However, in any subsequent use of *mary*, as drawn from this set, only her PERSON attributes will be seen. Class conformability is an important issue in ADAMS, but one which can be largely ignored when first learning the language. Reasonable usage will cause no problems, it is only there to detect and prevent unreasonable usage.

Elements of named sets need not themselves be named, because they can always be retrieved from the set. Rather than naming such elements we most often use code such as

```
        while (more_students)
              {
              printf ("Enter student name > ");
              scanf  ("%s", p_name);
              printf (" Enter student age > ");
              scanf  ("%d", &p_age);
              printf (" Enter student gpa > ");
              scanf  ("%f", &gpa);
<<            x  instantiates_a  STUDENT    >>
<<            x.name  <-  | p_name char* |  >>
<<            x.age   <-  | p_age  int   |  >>
<<            x.gpa   <-  | gpa    float |  >>
<<            insert  x  into  undergrad    >>
                 .
                 .
              }
```

to enter new "unnamed" elements into the data space. Note here that the symbolic name *gpa* denotes both a host variable, and an ADAMS attribute. This is permissible, because the ADAMS and host language name spaces are disjoint; however, it is not recommended.

An **enumerated set**, delimited by { ... }, is one whose elements have been explicitly numerated. Attributes are ADAMS elements, so one can have a set of attributes just as one can have any set of conformable elements. In fact, we have been implicitly using enumerated sets of attributes in our examples above. In the class declaration

```
<<    PERSON  isa  CLASS
            having  attrs = { name, age } >>
```

we used the enumerated set { *name* , *age* } to denote the set of associated attributes.

Similarly, in the **set assignment**

```
<<    friends  <-  { bill, mary, tom }    >>
```

*friends* denotes a set of just these 3 PERSON elements. Set assignment requires some discussion. If *friends* is an ADAMS variable, then it simply denotes the enumerated set on the right side. Suppose instead, that *friends* is an instantiated set of class PEOPLE, that is, of PERSON elements. First, the existing set denoted by the left hand side is made empty, then each of the elements of the right hand set is inserted into this set. Note, that a single ADAMS element may be a member of many different sets. Moreover, if one changes that element in anyway, say by reassigning a new attribute value, that change is reflected throughout all the sets to which it may belong. This is quite different from the relational model, in which a single tuple can only belong to one relation. An exact copy may belong to a different relation; but they are operationally regarded as distinct tuples, changing one will not affect the other.

The standard set operators, **union**, **intersection**, and **relative complement** (or set difference) are defined over all sets of conformable elements. Statements such as

```
<<    all_students   <-  undergrad  union  graduate   >>
<<    some_students  <-  all_students  intersect
                              { bill, mary, tom }     >>
<<    other_students <-  all_students  complement
                              some_students           >>
```

are common in ADAMS code. These operators may also be used in more general expressions, as we will see below.

A subset of the elements in an ADAMS set can be denoted by a **retrieval set**. Retrieval sets in ADAMS function in much the same way that the selection operator functions in the relational model. The general form of a retrieval set is:

{ <*ADAMS_var*> in <*base_set*> | <*predicate*> }

where the boolean selection <*predicate*> provides the criterion for deciding whether the element denoted by <*ADAMS_var*> belongs in the retrieval set. Retrieval sets can be quite complex. For now, it will be best to simply look at a few examples, and let the details wait until Section 4.1.3. The following are four representative retrieval sets:

```
{ x  in  undergrad | x.gpa > '3.0'  }
{ y  in  graduate  | y.name = 'Smith' and  y.age != '25' }
{ z  in  undergrad union graduate |
                  '17' < z.age <= '21' or z.gpa > '3.0'   }
{ x  in  { bill, mary, tom } |  x.age >= | p_age int |   }
```

The sense of each should be self-evident. We will only note that (a) the <*base_set*> from which the elements are to be retrieved can be any set expression, (b) logical conjunctives *and* and *or* behave as expected, (c) all literals are quoted, even numeric literals, (d) one may employ double ended range search in the natural way, and (e) host variables may be used in the predicate. Retrieval sets frequently occur in the right hand of set assignments, but they may employed in any set expression, just like any other set.

Iteration is important in any programming language. The *for* and *do* constructions, both of which loop over discrete sets, are familiar examples. Iteration is equally important in ADAMS. One can iterate over a set, appropriately using each element one at a time. The general structure of a **for_each statement** is:

```
<<      for_each <ADAMS_var> in <loop_set> do
          .
          .
          < any combination of ADAMS or host_language statements>
          .
          .
          >>
```

Notice that the terminating delimiter of the *for_each* statement may occur many lines removed from the beginning of the loop. Let us consider a simple example. To display the grade point average of all students, one might employ the code

```
<<    for_each  x  in  graduate union undergrad  do
<<          | p_name char* | <- x.name    >>
<<          | gpa    float | <- x.gpa     >>
            printf ("%5.3f - %s\n", gpa, p_name);
            >>
```

## 2.5. Maps

An ADAMS attribute is a single valued function, which given an element as argument, returns a single data value. An ADAMS **map** is a single valued function, which given an element as argument, returns a single ADAMS element. Because, a map is also an ADAMS element, it must belong to a class. For example,

```
<<    PERSON_MAP  isa  MAP
            with image PERSON >>
```

defines the class of all maps, or mappings, into the class PERSON. Because entire sets are also elements, one can have "set-valued" maps by declaring the image space to be a SET class, as in

```
<<    PEOPLE_MAP  isa  MAP
            with image PEOPLE >>
```

Like attributes, specific maps must be instantiated. If, for example, we wanted the students in our

running example to have an advisor, we could use the following code:

```
<<    advisor  instantiates_a  PERSON_MAP >>

<<    STUDENT  isa  PERSON
            having  attrs = { gpa }
            having  maps  = { advisor }   >>
```

Note that (a) we must have two separate associated sets, one of attributes and one of maps, because they are not comparable elements, and (b) the attributes and maps associated with a class declaration must be instantiated before the class itself. (This latter constraint, can be relaxed by employing a FORWARD construct discussed in Section 4.2.3; thereby allowing the definition of maps from a class back into itself.)

Given that the advisor map has been defined on elements of class STUDENT, one can now perform element assignments of the form

```
<<    mary.advisor  <-  bill  >>
```

or retrieve information regarding map images, as in

```
<<    | p_name char* |  <-  x.advisor.name      >>
<<    | p_age  int   |  <-  x.advisor.age >>
```

The more complex dot expressions on the right side of these assignments are worth noting. Assuming that *x* is an ADAMS variable denoting a STUDENT element, then *x.advisor* denotes his, or her, advisor of class PERSON, and so can be a valid argument to both the *name* and *age* attributes. Dot expressions, as they are called in ADAMS, are always evaluated from left to right. They can be arbitrarily long, and provide a mechanism for navigating through the ADAMS data space.

Readers familiar with the relational model will have noticed that there has been no mention of the relational join operator; and those familiar with the object-oriented model will be aware that we have not discussed sub-objects, only sub-classes. The map construct provides the analogue to both these important concepts. Using the object-oriented paradigm, the *advisor* of *x* can be regarded as a "sub-element" of the element *x*. The ADAMS dot expressions are precisely the same as those used in both C and C++.

Justifying the role of maps as a surrogate for the join operator is a bit more difficult. First, if both relations have the same schema, then the join operation is an intersection operation, which is explicitly provided by ADAMS.[2] Next, we would note that "a join is lossless, if the join attribute(s) constitute a key to either of the two argument relations", that is, at most one tuple in, say the second, relation can join with any tuple in the first relation. In the case of such lossless joins, the join attribute(s) function as the symbolic pointer of a many-to-one mapping. It is more accurate to say that the ADAMS map can be used to capture the relational join operation when it is being used to implement a lossless, symbolic pointer, join. Or equivalently, when an attribute functions as a *foreign key* in a relational schema, it is replaced by a map in ADAMS. General lossy joins can be duplicated by ADAMS code (c.f. Section 5.4), but are not supported directly by the language itself.

---

[2] One must be careful here. ADAMS intersection and relational intersection need not be the same. In the relational model, tuple identity is "value based", that is two tuples are regarded as the same, or equivalent, if they have identically the same attribute values. In ADAMS, element identity is independent the element's values; two distinct elements may have the same attribute values. ADAMS is an "object-based" language, because the unique identity of an element is established when it is instantiated.

## 2.6. Miscellaneous Details

In this section, we merely want to mention a few details that have either been glossed over for clarity in the preceding discussion, or omitted altogether.

First, all ADAMS reserved words and constructs are delimited by "white space", where blanks, tabs, and new lines are regarded as "white space". In addition, all commas in ADAMS are also regarded as "white space". We frequently include commas for program readability, as in enumerated sets, but they are never necessary.

Second, ADAMS is initiated, and terminated by two statements:

```
<<    open_ADAMS  job_id      >>
<<    close_ADAMS job_id      >>
```

which must be the first, and last, executed ADAMS statements, respectively. The *job_id* can be any string variable; however, while necessary for syntactic validity, it is not used in the current implementation.

Third, there are a number of pre-defined ADAMS classes, attributes, and maps. These are used so commonly that their definition at SYSTEM scope seems valuable. Some, we have used in the preceding examples; others will be used in the following examples in Section 3. These are:

Predefined classes:

| | |
|---|---|
| STRING_ATTR | - class of all string valued attributes |
| INTEGER_ATTR | - class of all integer valued attributes |
| REAL_ATTR | - class of all real valued attributes |
| ATTR_SET | - SET of ATTRIBUTE elements |
| MAP_SET | - SET of MAP elements |

Predefined attributes:

| | |
|---|---|
| name_of | - STRING_ATTR, the "name" of any element (if named) |
| class_of | - STRING_ATTR, the "class" of any element |
| element_class_of | - STRING_ATTR, the "class" of element in any set |
| image_of | - STRING_ATTR, the "image class" of any map |
| codomain_of | - STRING_ATTR, the "codomain" of any attribute |
| cardinality | - INTEGER_ATTR, the cardinality (size) of the set argument |

Predefined maps:

| | |
|---|---|
| attributes_of | - ATTR_SET, the set of all attributes associated with the class |
| maps_of | - MAP_SET, the set of all maps associated with the class |
| element_of | - class indeterminate, some (arbitrary) element of the set |

Finally, we must describe the preprocessing and compilation sequence in ADAMS. Any source file containing ADAMS statements must be suffixed with (`.src`). Suppose, for example, we have a source program called `test_program.src.` The command

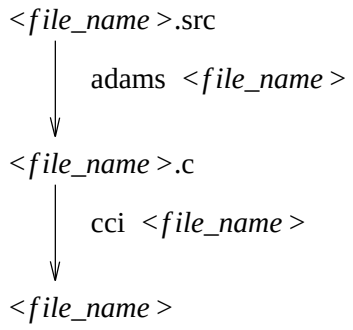
```
adams  test_program
```

will convert this file into one named `test_program.c.` Note that the `.src` suffix was not used in the adams preprocessor command. To do so will result in an error. The reader should take the time to examine this file, in which ADAMS statements are replaced with appropriate host language statements.

Next, the command

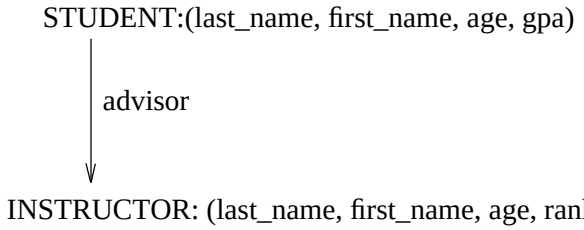
```
cci  test_program
```

is issued, which compiles the program, now completely in the host language, and creates an executable file that has been linked to modules in the ADAMS run-time system. One can now execute the `test_program` file. Schematically, the steps to create an executable program containing ADAMS statements are



## 3. Simple Examples of ADAMS Code

We can put together the fragments of code developed in the preceding section, to define a simple school structure in the data space. Initially, it will only consist of STUDENTS and FACULTY, together with a single many-to-one relationship *advisor* between them. The E-R diagram [Che76] would look like:




Execution of the following code creates the desired attributes, classes, and maps, which are entered into the user's name space.

```
main ()
       /*
       **  This program creates a simple school database
       **  consisting of STUDENTS and FACULTY, with a
       **  single 'advisor' relationship.
       */
       {
<<     open_ADAMS  job_id  >>

                     /* First, create appropriate  */
                     /*        attributes          */
<<     last_name   instantiates_a  STRING_ATTR    >>
<<     first_name  instantiates_a  STRING_ATTR    >>
<<     age         instantiates_a  INTEGER_ATTR   >>
<<     gpa         instantiates_a  REAL_ATTR      >>
<<     dept        instantiates_a  STRING_ATTR    >>
<<     rank        instantiates_a  STRING_ATTR    >>

                     /* Next,  declare element classes */
<<     PERSON     isa  CLASS
              having attrs = { first_name, last_name, age }  >>
<<     INSTRUCTOR isa  PERSON
              having attrs = { dept, rank }     >>

<<     INSTRUCTOR_MAP isa  MAP
              with image  INSTRUCTOR                  >>
<<     advisor    instantiates_a  INSTRUCTOR_MAP      >>

<<     STUDENT    isa  PERSON
              having attrs = { gpa }
              having maps  = { advisor }        >>

<<     FACULTY    isa  SET
              of  INSTRUCTOR  elements          >>
<<     STUDENTS   isa  SET
              of  STUDENT  elements             >>

                     /* Finally, create three sets  */
                     /*          in the data space  */
<<     faculty    instantiates_a  FACULTY      >>
<<     undergrad  instantiates_a  STUDENTS     >>
<<     graduate   instantiates_a  STUDENTS     >>
```

```
<<     close_ADAMS  job_id >>
       }
```

Definition of a School Database Structure
Figure 3-1.

Readily, in a real database one would have many more attributes associated with both students and faculty

Next, we must have data entry and display routines. Note that the ADAMS language, itself, provides neither. Knowing the attributes of STUDENT elements, it is possible to write code that is carefully tailored to the expected data entry situation. However, the constructs of ADAMS also facilitate the coding of rather generic data entry routines, which need not know the actual names of all the associated attributes. The following program searches the class declaration to obtain the set of associated attributes.

```
main ()
       /*
       **  This program interactively, accepts
       **  student data from the user, for entry
       **  into the data space.
       */
       {
       char       a_name[80], a_value[80],
                  response[15];
<<     ADAMS_var  x, a, view      >>

<<     open_ADAMS  job_id  >>

<<     view  <-  STUDENT.attributes_of  >>
       *response = 'y';
       while (*response == 'y')
              {
<<            x  instantiates_a  STUDENT >>
<<            for_each  a  in  view  do
<<                   | a_name char* |  <-  a.name_of  >>
                     printf ("Enter %s > ", a_name);
                     scanf  ("%s",  a_value);

<<                   x.a  <-  | a_value char* | >>
                     >>
              printf ("graduate or undergrad ? > ");
              scanf  ("%s", response);
              if (*response == 'g')
                     {
<<                   insert  x  into  graduate  >>
                     }
              else
                     {
<<                   insert  x  into  undergrad >>
                     }
              printf ("More students to be entered? (y/n) > ");
              scanf  ("%s", response);
              }

<<     close_ADAMS  job_id >>
       }
```

A Student Data Entry Program
Figure 3-2.

With very few modifications, e.g. changing the set assignment to get the attributes of FACULTY instead of STUDENT, the same code can be used to enter faculty data. The facility for writing still more generic programs, which do not have any "hard wired" class or attribute names is an important aspect of ADAMS. ADAMS supports dynamic class modification, or schema evolution, in which the set of attributes associated with any class can be enlarged, or shrunk, as experience with the database dictates. However, to make this data entry program completely generic would require several constructs which will be discussed in the Section 4.

Notice the value assignment, *x.a* <- | *a_value char** |. One student attribute, *age*, is integer; another attribute, *gpa*, is real. But, the code in this generic loop can't know that. Consequently, all data must be accepted as a character string. If conversion is needed, it will occur when the value assignment is executed.

A more interesting, and complex, program is that which assigns advisors to students. If *s* denotes the student element, and *f* denotes the faculty element, then it is sufficient to simply execute the single ADAMS statement

```
<<    s.advisor  <-  f    >>
```

The problem, as shown below, lies in determining which element *s* denotes the correct student, and which element *f* denotes the correct faculty member.

```
main ()
        /*
        **  This code assigns faculty advisors to students
        */
        {
        char       l_name[80], f_name[80], dept[8], rank[15],
                   a_value[80], response[15];
        int        size;
<<      ADAMS_var  a, s, f, s_view, f_view, search_set, result    >>

<<      open_ADAMS  job_id    >>

                           /* attributes to disambiguate */
                           /* student, faculty identity  */
<<      s_view  <-  { first_name, age }           >>
<<      f_view  <-  { first_name, dept, rank }    >>

        *response = 'y';
        while (*response == 'y')
            {              /* First get the student element */
            printf ("Enter student's last name > ");
            scanf  ("%s", l_name);
            printf ("graduate or undergrad ? > ");
            scanf  ("%s", response);
            if (*response == 'g')
                {
<<              search_set  <-  graduate     >>
                }
            else
                {
<<              search_set  <-  undergrad    >>
                }
<<          result  <-  { s in search_set | s.last_name = | l_name char* | } >>
<<          | size int |  <-  result.cardinality    >>
            if (size == 1)
                {
<<              s  <-  result.element_of    >>
                }
            else if (size > 1)
```

```
                {           /* pick correct student */
                printf ("%d students have last name '%s'\n",
                        size, l_name);
                printf ("Pick correct one:\n");
<<              for_each  s  in  result  do
                    printf ("\t");
<<                  for_each  a  in  s_view  do
<<                      | a_value char* |  <- s.a    >>
                        printf ("%s ", a_value);
                        >>
                    printf (" ? (y/n) > ");
                    scanf  ("%s", response);
                    if (*response == 'y')
                        {
<<                      exit_loop    >>
                        }
                    >>
                }
            else
                {
                printf ("No student with last name '%s'\n", l_name);
                goto  continue;
                }

                            /* Now, get correct faculty element */
            printf ("Enter advisors's last name > ");
            scanf  ("%s", l_name);
<<          result  <-  { f in faculty | f.last_name = | l_name char* | } >>
<<          | size int |  <-  result.cardinality    >>
            if (size == 1)
                {
<<              f  <-  result.element_of    >>
                }
            else if (size > 1)
                {               /* pick correct professor */
                printf ("%d faculty have last name '%s'\n",
                        size, l_name);
                printf ("Pick correct one:\n");
<<              for_each  f  in  result  do
                    printf ("\t");
<<                  for_each  a  in  f_view  do
<<                      | a_value char* |  <- f.a    >>
                        printf ("%s ", a_value);
                        >>
                    printf (" ? (y/n) > ");
                    scanf  ("%s", response);
                    if (*response == 'y')
                        {
<<                      exit_loop    >>
                        }
                    >>
                }
            else
                {
                printf ("No faculty with last name '%s'\n", l_name);
                goto  continue;
                }
                            /* assign advisor */
<<          s.advisor  <-  f    >>

continue:
            printf ("More advisors to be assigned? (y/n) > ");
            scanf  ("%s", response);
```

```
            }

<<      close_ADAMS  job_id    >>
        }
```

Assigning Elements to the *advisor* Map
Figure 3-3.

This program illustrates a weakness of ADAMS, or one of its greatest strengths, depending on one's point of view. In a relational database, one simply enters the advisor's name in the student tuple to provide a "foreign key". It is much simpler. But, relational databases are also bedeviled by what is called the "referential integrity problem". If the advisor's name is misspelled, or a variant of the first name is used, the join operation will not be able to pick up that advisors attributes, or may join with some other advisor! Moreover, such discrepancies are very hard to discover.

Because ADAMS must establish the element identity, it forces the application code to take appropriate steps to assure it, and at the same time provides mechanisms to accomplish it. Referential integrity is a very much smaller problem in ADAMS databases than it is in relational databases.

Finally, this program shows how host language code and ADAMS statements can be combined. ADAMS was never designed to be "user friendly" in the sense of being easy for non-programmers to use. It isn't. It was designed to facilitate easy access to a persistent data space by experienced programmers, who can then construct appropriate user interfaces for others to use.

It is possible to write generic programs to display data in much the same manner as the data entry program, c.f. the *show_set* procedure of Section 4.2.14; but they are seldom completely satisfactory. Typically, one wants to format the output in some appropriate way. Shown below is a vanilla display program which lists students in the database.

```
main ()
        /*
        **  Display a list of all students,
        **    together with their gpa's, and advisors.
        */
        {
        char        f_name[80], l_name[80], rank[80];
        float       s_gpa;
<<      ADAMS_var   s >>

<<      open_ADAMS   job_id >>
        printf ("Undergraduate Students:\n");
<<      for_each  s  in  undergrad  do
<<              | f_name char* |  <-  s.first_name      >>
<<              | l_name char* |  <-  s.last_name >>
                printf ("\t%s, %s", l_name, f_name);
<<              | s_gpa  float |  <-  s.gpa       >>
<<              | l_name char* |  <-  s.advisor.last_name      >>
                printf ("\t%5.3f, advisor: %s\n",
                            s_gpa, l_name);
                >>

<<      close_ADAMS  job_id >>
        }
```

A simple listing display
Figure 3-4.

which might generate

```
Undergraduate Students:
       Scott, Mary     3.600, advisor: Pfaltz
       White, Bill     2.561, advisor: Grimshaw
       Black, Henry    3.873, advisor:
       Scott, John     2.834, advisor: French
       Nix, William    3.208, advisor: Grimshaw
       Talley, Mike    3.350, advisor:
         .       .       .       .
         .       .       .       .
```

Sample output from the listing program
Figure 3-5.

We can make two observations. First, two of the students have no advisor; so the element designator *s.advisor* that is the argument for the *last_name* attribute must be NULL. ADAMS is smart enough to handle this correctly. Second, although the program description says it will list *all* students, the code actually only lists *undergrad* students. Duplication of the loop, but over *graduate* will do this. But, wouldn't you expect some sort of procedure facility to avoid this kind of repetitive code? It exists, and will be covered in Sections 4.2.14-16.

## 4. Formal Description of the ADAMS Language

ADAMS is a language for describing and accessing persistent data independent of any particular programming language. At the same time, it provides an interface to most popular programming languages, so that applications and user interface modules may be developed and coded in those idioms. The element concept is fundamental in ADAMS; and an underlying principle of the language is that — although we have four different kinds of elements: garden variety elements, attribute elements, map elements, and set elements — all elements will be treated identically. For example, all elements must belong to a class declared by an *isa* statement, and are created by *instantiates_a* statements; any element class can have associated attributes and maps; the naming conventions for all kinds of elements are identical; and one can construct sets of any kind of element. Limiting the language to just these four generic kinds of elements, with a uniform treatment of all, has been a major design principle of ADAMS.

Like most programming languages, ADAMS consists of statements comprised of expressions. However, instead of executable arithmetic and boolean expressions, ADAMS is based on three kinds of *designational* expressions: (1) element designators, (2) value designators, and (3) set designators. The typical expression in an ADAMS statement designates some relevant portion of the persistent data space, hence this terminology.

In this section, we first examine each of these designational expressions in detail. Then, we discuss each of the ADAMS statements in turn. This is a description of the complete ADAMS language. But, not all features have been implemented in the current version. Currently unimplemented features will be marked by a dagger †, or otherwise indicated. The sample code in the preceding Section 3, and in the following Section 5, represent only those features of ADAMS which are currently implemented. All code in those sections has been compiled and executed.

### 4.1. ADAMS Designational Expressions

A **designational expression** denotes a unique ADAMS element or a unique value, where the latter is simply assumed to be a bit string. There are element designators, value designators, and set designators.

#### 4.1.1. Element Designators

There are three kinds of **element designator**, or <*elem_desig*>, in ADAMS: ADAMS names, ADAMS variables, and dot expressions. Every **ADAMS name** is entered into the name space at the time the element it denotes is instantiated, and so long as it remains in the name space, is bound to that particular element and can denote no other. ADAMS names that are used to denote elements may be either actual names, var names, or subscripted names.

An **actual name** consists of character segments, each of which must contain an alphabetic character, that are separated by underscores '_'.

```
mary
3x3_matrix
attr_view
```

are actual names. If the actual name is known to have a particular visibility scope in the name space, that scope may be prefixed, as in

```
TASK  attr_view
```

thereby creating a **scoped name**. See Section 6.1 for more detail regarding the use of scoped names.

A **var name** is actually a string variable of the host language whose current value is to be treated as an actual name. The variable name is preceded by the reserved word **var**, as in

```
var set_name
```

where *set_name* has been declared to be of type *char** in the host code. The use of var names provides the capability of manipulating ADAMS names within the host language, but since the current value of the variable must be compared at run-time with the name space to ensure validity, execution is slower.

A **subscripted name** is an actual name followed by one, or more, non-negative, integer subscripts delimited by [ ... ]. A **subscript** may be an integer, an actual name (which in this case is assumed to be a host variable name), or a host language expression delimited by @...@,[3] which will evaluate to a non-negative integer. The subscript list may be optionally separated by commas.

```
particle[x , y, z]
point[ 3 j 17 ]
particle [ 23, @i+j*3@, n ]
```

are subscripted names. Attribute and map names may be subscripted. Identical names, but with different subscript values, denote different elements. Subscripted element names must be declared when the element is instantiated (c.f. element instantiation statement), and because they really denote a family of elements, subsequent element instantiation is somewhat different from elements instantiated with unsubscripted ADAMS names.

Any ADAMS name may be declared to be an **ADAMS variable** (c.f. ADAMS_var statement, 4.2.1). Such variable names are not entered into the name space, and are not bound to a single ADAMS element. ADAMS variables may be bound to specific elements by either element instantiation statements or by element assignment statements. The binding persists only until rebound to another element, or until termination of the program. ADAMS variables may be subscripted.†[4]

Finally, elements may be denoted by dot expressions. A **dot expression** of the form <*elem_desig*> . <*map_desig*> denotes that element resulting from evaluating the map denoted by the second <*map_desig*> term using the first <*elem_desig*> term as argument. Dot expressions are evaluated from left to right, so <*elem_desig*> may itself be a dot expression

```
mary.advisor
x.father.father
point[i, j, k].nbhrs
```

are valid dot expressions that denote elements.

### 4.1.2. Value Designators

A **value** is a bit string of indeterminate length that has been represented in the data space. Such values may be denoted by either literal expressions or by dot expressions.

A **dot expression** of the form <*elem_desig*> . <*attr_desig*> denotes that value returned by the attribute function <*attr_desig*>, when given the <*elem_desig*> as element argument. Because these values belong to the codomain (or range, or image space) of the attribute function, we call the set of all possible values that can be designated by such a dot expression, a **codomain**.

A **literal** in ADAMS consists of any string of characters enclosed by either single quote, '...', delimiters or double quote, "...", delimiters. The enclosed characters must belong to the appropriate codomain.

---

[3] A much better subscript expression delimiter will be included in the next version.

†[4] Subscripted ADAMS variables are not currently implemented, however they have been in earlier versions, and will be added shortly.

It is convenient to treat **host variable** expressions of the form

| <*variable_name*> <*type*> <*length*> |

as if they were <*value_desig*>'s, even though semantically they are quite different. Value designators denote values in the persistent data space, whereas host variable expressions denote values in the program's data space. However, they perform the same function syntactically; they denote values. Because ADAMS does not know the program's data space, the variable name must be delimited in ADAMS statements with |...| and the variable <*type*> explicitly given. Because ADAMS data consists only of bit strings of indeterminate length, a <*length*> designator†[5] may optionally be specified to prevent overwriting string variables in assignment statements.

### 4.1.3. Set Designators

Since ADAMS sets are themselves elements, any element designator described in Section 4.1.1 above may denote a set. However, there are four more designational expressions which may only be used denote set elements. The first three of these expressions, (1) enumerated sets, (2) set expressions, and (3) retrieval sets, are derived from corresponding expressions used in mathematics to describe sets. The sets that are so denoted are non-persistent, unless assigned to, or otherwise included in, a persistent set. In mathematics, it is customary to require that all elements of a set belong to the same class. ADAMS enforces this same constraint. In the following discussion of sets, we let the *cose* of a set denote the class of the set elements comprising the set. Note that specific elements in the set may belong to a subclass of the *cose*, that is be **conformable** to the *cose*; but all will be interpreted as belonging to the *cose*.

An **enumerated set** is any list of element designators delimited by {...}. The <elem_desig> of the list may optionally be separated by commas. To facilitate the enumeration of subscripted elements, one may denote a range of subscript values. A **range subscript** has the form <*subscript_value*>..<*subscript_value*>, as in

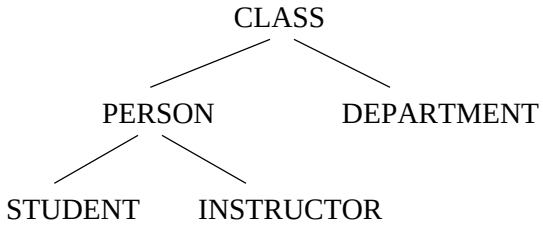
```
{ mary, bill, student[ 3 .. 6 ] }
```

in lieu of

```
{ mary, bill, student[3], student[4], student[5], student[6] }
```

Range subscripts may be used only in enumerated sets.

The *cose* of the enumerated set is the least upper bound of the classes of each of the individual elements. For example, given the class structure




one can form an enumerated set composed of elements of both STUDENT and INSTRUCTOR classes. However, the *cose* of the enumerated set will be PERSON. This is important, because even though it may be known that a particular element belongs to the class STUDENT, only those attributes, maps, and properties associated with its super class PERSON may be employed — so long as that element is referenced as a member of the enumerated set. Similarly, one may construct an enumerated set consisting of PERSON and DEPARTMENT elements; the *cose* of the enumerated set will be simply CLASS. But no attributes or maps can be employed, because this generic class has none.

---

†[5] In the current preprocessor, <*length*> must be a literal integer.

Attempts to create an enumerated set of both attribute and map elements will result in an ADAMS error, because the classes ATTRIBUTE and MAP have no upper bound. Similarly, if one element of the enumerated set is itself a SET, then all must be of class SET. ADAMS does not yet enforce conformability on distinct SET classes.

A **set expression**, or <*set_expr*>, is any <*set_desig*> or the result of applying one of the three binary set operators *union*, *intersect*, and *complement* to its operands. The class of the elements comprising these sets, or *cose*, is dependent on the particular operation.

(1) **union**: The set expression <*set_desig*$_1$> *union* <*set_desig*$_2$> denotes those elements that are members of either <*set_desig*$_1$> or <*set_desig*$_2$>. The *cose* of the designated set is the least upper bound of the *cose* of the operand sets.

(2) **intersection**: The set expression <*set_desig*$_1$> *intersect* <*set_desig*$_2$> denotes those elements that are members of both <*set_desig*$_1$> and <*set_desig*$_2$>. The *cose* of the designated set is the greatest lower bound of the *cose* of the operand sets.†[6]

(3) **complement**: The set expression <*set_desig*$_1$> *complement* <*set_desig*$_2$> (also called set difference) denotes those elements that are members of <*set_desig*$_1$> but not <*set_desig*$_2$>. The *cose* of the designated set is the *cose* of the first operand.

If a set expression contains more than one operator, their virtual execution is left to right. Set expressions may be parenthesized.†[7]

Sets may be denoted by rule, or by a predicate whose evaluation governs membership within the set. Such a set is called a **retrieval set** in ADAMS. A retrieval set has the general form

{ <*ADAMS_var*> in <*set_expr*> | <*predicate*> }

where the <*predicate*> is any boolean predicate in which the <*ADAMS_var*> is a free variable. Conceptually, each element in <*set_expr*> is, in turn, denoted by <*ADAMS_var*>, and <*predicate*> is evaluated to determine if this element belongs in the set. The *cose* of the retrieval set is the *cose* of <*set_expr*>.

The structure of the <*predicate*> supports <*terms*>'s which may be combined with a logical **and** to form <*conjunct*>'s, which may be combined with a logical **or** to form <*disjunct*>'s. This follows the usual pattern of boolean predicates found in most programming languages — the reader is referred to the syntax in Section 7 for details.

Of most interest for this discussion is the different kinds of <*term*> that are available in ADAMS. A **term** may be either (1) an element comparison, (2) a value comparison, or (3) a quantified term. An **element comparison** of the form

<*elem_desig*$_1$> <*elem_comp*> <*elem_desig*$_2$>

where only = or != are valid <*elem_comp*>, is true if the two <*elem_desig*>'s do (do not) denote the same element.

A **value comparison** may have either of two forms:

<*value_desig*$_1$> <*comparator*> <*value_desig*$_2$>

or

<*value_desig*$_1$> <*comparator*> <*dot_expr*> <*comparator*> <*value_desig*$_2$>,

where <*comparator*> may be any of =, !=, <, <=, >, or >=. The latter is a convenient way of

---

†[6] The current preprocessor requires that *cose* of the operands be the same class, or at least one a subclass of the other.

†[7] Parenthesized set expressions are not yet recognized by the preprocessor.

denoting double ended range searches.†[8] An example of a retrieval set using both these kind of terms is

```
{ x in graduate | '3.0' <= x.gpa < | max_gpa float |
                  and  x.advisor = mary.advisor      }
```

which denotes all *graduate* students whose *gpa* is between 3.0 and *max_gpa*, and who also have the same advisor as *mary*.

The third, **quantified term** is not yet implemented. It may have the form of either

( exists <*bound_var*> in <*set_expr*> ) [ <*predicate*> ]

or

( all <*bound_var*> in <*set_expr*> ) [ <*predicate*> ]

which is true if at least one (or all) elements in <*set_expr*> satisfy the predicate.

We have called enumerated sets, retrieval sets, and the results of set operations, set *designators* because that is precisely what they are. They are not set constructors. New sets may be added to the data space only by explicit instantiation. The usual procedure is to instantiate a set (c.f. Section 4.2.6 below) either as a named, or unnamed set, and then assign the elements denoted by a set designator to it, as in

`<<` <*set_name*> `<-` <*set_desig*> `>>`

The final set designator has no mathematical equivalent. Sets that have been associated with element classes may be denoted using the **association operator**, ->, in expressions of the form:

<*class_name*> -> <*synonym*>

(c.f. class declaration statement, 4.2.3).

## 4.2. ADAMS Statements

We now discuss each of the ADAMS statements, one by one. These statements provide the mechanism by which the designational expressions of ADAMS are included in a host programming language. All ADAMS statements are delimited by << ... >>.[9] This presentation will begin with a syntactic description of statement, followed by a discussion of possible variations, or options, when available. The operational semantics may be discussed, if appropriate; but, except for one case, not the method of implementation. This is a language. It is quite possible for different versions to implement it differently.

Any ADAMS statement can fail in execution. For example, an attempt to instantiate a new element may fail because the name already exists in the name space; or an attempt to insert an element into a set may fail because the class of the element does not conform to the *cose* of the set. The preprocessor attempts to detect most such failure situations and flag them as errors. But, static checking cannot detect all possible failure situations, particularly when the host program employs `var` names to manipulate the name space. For many uncertain situations the preprocessor will issue a warning. When an ADAMS statement fails, a global variable `_ADAMS_FAIL` is set to a non-negative error code.†[10] This can be tested at any time in the host_code, and reset after

---

†[8] Currently, only the comparators < and <= may be used — in any combination. The comparators > and >= will be implemented in the next version.

[9] Both << and >> are legal operators in C and C++ code. The ADAMS preprocessor can, in most cases, distinguish their use.

†[10] In the current implementation, setting `_ADAMS_FAIL` will, in most cases, be immediately followed by a system abort which gracefully shuts down the ADAMS system.

appropriate action has been taken.

### 4.2.1. ADAMS_var Statement

An **ADAMS_var** statement of the form:

```
<<    ADAMS_var  x, y, temp_set, attr     >>
```

establishes that the symbolic names *x*, *y*, *temp_set*, and *attr* are to be regarded as ADAMS variables which can denote any element of any class. This statement, which only provides information for the preprocessor, is not executable and may appear anywhere in the host code. There may be multiple ADAMS_var statements in any source file.

Because, the preprocessor does not parse host language code, ADAMS_var's are not restricted to a single procedure module, as they ought to be. The declaration is valid for the entire source code file, and hence the declaration defines such ADAMS variables *globally*. Redeclaration of an ADAMS variable generates a warning, but no error.

### 4.2.2. Assignment Statement

Assignment in ADAMS is denoted by the **assignment operator**, <-. It may used in three different assignment statements, all of which have similar syntax, but which have different semantics.

A **value assignment** has the form:

<< <*value_desig*$_{left}$> <- <*value_desig*$_{right}$> >>

where the left side may be either a dot expression or a host variable expression, and the right side may be either of these, or a literal. The value of the right side is assigned as the value of the left side. At least one side of a value assignment must be a dot expression, so ADAMS can determine the attribute and its associated codomain.

In assignments to, and from, host variables such as

```
<<    | attr_value  char*  10 |  <-  x.attr     >>
<<    x.attr  <-  | attr_value  char* |         >>
```

coercion may take place when *attr* is a numeric attribute. ADAMS handles this correctly.

An **element assignment** has the form:

<< <*elem_desig*$_{left}$> <- <*elem_desig*$_{right}$> >>

where the left side must be either an ADAMS variable or a dot expression, and the right side may be any <*elem_desig*>. If the left side is an <*ADAMS_var*> then this assignment makes the ADAMS variable denote the same element as the right side, and the class of the ADAMS variable becomes that of the right side element. If the left side is a dot expression of the form <*elem_desig*>.<*map_desig*>, then the image of <*elem_desig*> under <*map_desig*> is changed to the right side element. The class of the right side element must be conformable to the image space of the <*map_desig*>. In both cases, the assignment alters what the left side denotes. For this reason, the left side <*elem_desig*> cannot be an ADAMS name, because ADAMS names are invariantly bound to elements when instantiated. The exception to this is the case when the left side is a set name, in which case the assignment is interpreted as a set assignment described below.

A **set assignment** has the form:

<< <*set_desig*$_{left}$> <- <*set_desig*$_{right}$> >>

where the left side may be either a <*set_name*> or a dot expression mapping into a SET class, and the right side may be any <*set_desig*>. The *cose* of the right set must be conformable to the *cose* of the left set. Set assignment employs shallow copy semantics. All existing elements of

the left set (if any) are first removed, then every element in the right set is inserted into the left set.

A special set literal, NULLSET, may be used in set assignment to empty a set. Mathematically, NULLSET denotes ∅ and its *cose* is conformable to any set. However, beware of assigning NULLSET to an ADAMS variable, as in:

```
<<    x  <-  graduate          >>
      .
      .
<<    x  <-  NULLSET           >>
```

Rather than making *x* now denote a null set, it empties the set that *x* currently denotes (in this case *graduate*). To have *x* denote a new null set, one simply instantiates the new set with

```
<<    x  instantiates_a  <set_class>,
            scope is LOCAL          >>
```

### 4.2.3. Class Declaration Statement

All **class declaration** statements have the general form:

```
<<    <class_name>  isa  <super_class_list>
            <class_decl_options>          >>
```

where the <*super_class_list*> may be either of the four generic element classes, CLASS, ATTRIBUTE, MAP, or SET; or the name of any class already defined in the name space; or a list of such classes, conjoined by **and**. The scope of all classes named in the <*super_class_list*> must be of the same, or higher, than the declared scope of the class declaration. In general, no named component of an ADAMS declaration may have lower scope than the declaration itself.[11] The new class automatically inherits the properties of *all* these super classes. For example, if STUDENT and INSTRUCTOR are subclasses of PERSON as defined in Section 2.5, then

```
<<    T_A  isa  STUDENT and INSTRUCTOR ...      >>
```

defines a class of teaching assistant elements which have properties of both students and instructors, and may be used in the roles of either. This is called **multiple inheritance**.

We will note that classes may only be denoted by name (there is no general <*class_desig*> analogous to <*elem_desig*>), so all classes must be explicitly named in the name space. An optional <*scope_clause*> of the form

```
scope is [ LOCAL | USER | TASK | SYSTEM ],
```

allows insertion of the <*class_name*> at any level of the name space hierarchy.

The real interest with respect to class declaration lies in the various possible <*class_decl_options*>. Any class may have one, or more, associated sets of maps or attributes declared by an **association clause** of the form:

```
having <synonym> = <set_expr>†[12]  .
```

where the <*synonym*> = phrase is optional.

---

[11] Due to a design defect in the preprocessor, it cannot keep track of the scopes of newly declared classes and instantiations. Consequently, it issues many patently unnecessary warnings.

†[12] The current preprocessor only accepts <*set_desig*> for an associated set.

```
<<    PERSON  isa  CLASS
            having  { name, age }          >>
<<    STUDENT isa  PERSON
            having  attrs = { gpa },
            having  maps  = { advisor }    >>
```

Such associated sets are ADAMS elements and may be dynamically modified by element insertion, deletion, or set assignment using the association operator, ->, to associate set synonyms with class declarations, as in PERSON->*attrs*, STUDENT->*attrs*, or STUDENT->*maps*. Association set **synonyms** must be distinct within any single class declaration, but may be repeated in separate declarations since any ambiguity will be resolved by association with the <*class_name*>.

Class membership may be restricted by means of a **restriction clause**†[13] of the form:

```
provided [ <predicate> ]
```

where <*predicate*>, delimited by [...], has a single free variable of the form *@x*. For example, one may declare

```
<<    ADULT  isa  PERSON
            provided [ @x.age >= 21 ]      >>
```

to restrict membership[14] in the class ADULT to instantiated elements whose *age* attribute has value ≥ 21. The free variable @x (or any single letter preceded by @) is interpreted to mean "any element", or the "current element". The semantics of the restriction clause must be discussed. When an element is first instantiated, as in

```
<<    x  instantiates_a  ADULT       >>
```

no *age* has been assigned, so it cannot possibly satisfy the restriction clause. Test of the restriction clause is deferred until the smallest transaction unit enclosing the instantiation statement terminates. If the restriction clause is not satisfied, both the statement and the entire transaction fails; no change is made to the persistent data space. If the superclass(es) of a class have restriction predicates, these are inherited by any subclass and their conjunction must be satisfied.

SET, ATTRIBUTE, and MAP class declarations each require an additional clause. Set declarations must declare the class of the set elements, or *cose*, with the clause

```
of <class_name> elements .
```

Attribute and map declarations must declare the image space of the function with the clause

```
with image <codomain_name>
```

or

```
with image <class_name>.
```

The need to provide the <*class_name*> of the image class in map declarations leads to problems in certain kinds of recursive class declarations. For example, suppose we want to define a class called PART, which has a map *is_part_of* into the PART class itself. The class declaration, PART, must precede the map class declaration; and the *is_part_of* map must be instantiated before the class declaration. To resolve this, ADAMS provides a *FORWARD* option in class declaration. The declaration

```
<<    PART  isa  CLASS,  FORWARD    >>
```

---

†[13] Restriction clauses have not been implemented in the current version.

[14] We would emphasize that membership in a class is very different from membership in an instantiated set. An ADAMS element can belong to only one class; but it can be a member in multiple sets.

only enters the <*class_name*>, PART, together with its super class(es) in the name space, and asserts that the complete declaration will be provided at some later time. We can now write

```
<<    PART_MAP  isa  MAP,
            with image  PART        >>
<<    is_part_of  instantiates_a  PART_MAP      >>
```

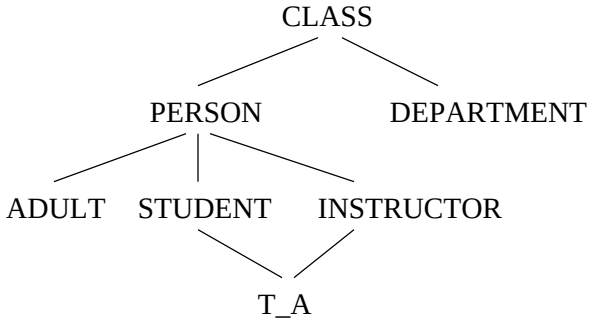
because the <*class_name*> has been defined; and then complete the PART declaration with

```
<<    PART  isa  CLASS,
            having attrs = { part_name, ..., cost }
            having maps  = { is_part_of }       >>
```

See Section 5.1 for an application requiring the FORWARD option.

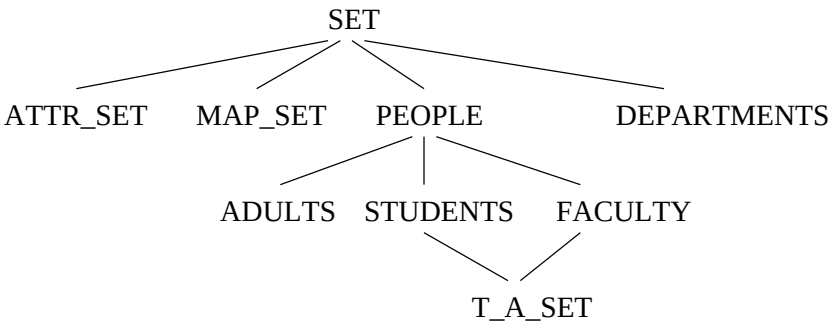
### 4.2.4. Class Inheritance Semi-lattices

The structure of classes and their inheritance properties may be diagrammed as a tree, as in the figure of Section 4.1.3; or if multiple inheritance is supported as a semi-lattice. For example, the classes we have defined so far would give rise to the semi-lattice



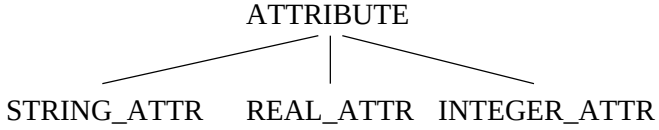

Readily, one can develop rather complex class semi-lattices. The concept of conformability, and *cose*, are all based on this class semi-lattice.

Since, for any class of element, one may have a set of these elements, the CLASS semi-lattice induces a corresponding SET class semi-lattice of the form




However, as can be seen, the SET semi-lattice need not be isomorphic to the CLASS semi-lattice. For more details regarding CLASS semi-lattices and their induced SET semi-lattices, see [Pfa88].

In fact there are four class semi-lattices. Besides the CLASS and SET semi-lattices illustrated above, we also have ATTRIBUTE and MAP semi-lattices. But, these tend to be rather uninteresting, shallow trees as in



The semantics of inheritance in the SET, ATTRIBUTE, or MAP hierarchies is not well understood at this time. It is one of important open research issues raised by ADAMS.

### 4.2.5. Codomain Declaration Statement

A **codomain declaration** statement is of the form:

```
<<    <codomain_name>  isa  CODOMAIN
            consisting of #<lex_expr>#>>
```

where the <*lex_expr*>, delimited by #...#, is a *lex* expression [LeS75] that designates the regular set comprising the codomain. In effect, the <*lex_expr*> specifies what literal strings will be regarded as well-formed values in that codomain. A preprocessor option can be set so that ADAMS will enforce codomain consistency for all values in every codomain.†[15]

To ADAMS, all values in its data space are only bit strings of indeterminate length, however it requires some additional information in order to present these values to the host program in their expected form. The optional **treat_as** clause may included to specify that the value will be treated as *alphanumeric*, *numeric*, or *raw*†[16] by the applications code. The default treat_as usage is *alphanumeric*, or simply character string.

One may optionally define what value to return if an attribute function <*elem_desig*>.<*attr_desig*> is undefined for that element with a **udf** clause of the form:

```
udf = '<literal>'
```

The default *udf* value is an alphanumeric string of no characters.

For example, the declaration of the REAL codomain that is predefined in ADAMS is:

```
<<    REAL  isa  CODOMAIN
            consisting of #[-|+]?[0-9]*.[0-9]*#
            udf = '-999999999.0',
            treat_as numeric,
            scope is SYSTEM              >>
```

### 4.2.6. Element Instantiation Statement

All elements are instantiated by the statements of the form:

```
<<    <actual_name>  instantiates_a  <class_name>    >>

<<    <ADAMS_var>  instantiates_a  <class_name>    >>
```

or

```
<<    <actual_name>[<subscript_count>] instantiates_a <class_name> >>
```

In the first case, the <*actual_name*> is entered into the name space and bound to the instantiated element. In the second case, the <*ADAMS_var*> denotes the instantiated element, but nothing is entered into the name space.

In the third case, <*subscript_count*> simply denotes a sequence of asterisks, which are white space delimited — often by commas, as in

```
<<    point[*,*,*] instantiates_a  POINT  >> .
```

What has been instantiated is a family of triply subscripted element names, rather than any

---

†[15] Enforcement of codomain consistency, has not been implemented in the current version, although stubs to invoke an appropriate *lex* routine exist.

†[16] The treat_as term *raw* is semantically meaningless in the current version, and is treated as *alphanumeric*.

particular element in the family. All subscripts must be non-negative integers, but there is no upper bound declared on this family. We must emphasize that this statement does not instantiate an array structure in the usual sense of programming languages, only a family of subscripted names, c.f. [PfF90]. Any subsequent reference to *point* followed by three non-negative subscripts will be assumed to denote a POINT element. Actual instantiation of this element will occur when it is first used. For example, in the assignment

```
<<    point[3,0,7].nbhrs <- { point[3,1,4], point[2,5,2] } >>
```

if *point*[3,0,17] had not be previously instantiated, it will be as a side effect of the assignment and the enumerated set assigned as its *nbhrs* map image. Although a subscripted instantiation does not create a vector, or array, structure in the data space, it may be used to represent such structures as described in Section 5.2.

All instantiated elements are by default persistent, and if named, with their element name in the USER name space. The optional scope clause may be used to enter their name in other levels of the hierarchical name space. To instantiate a temporary, non-persistent element which will vanish on termination of the program, it must be given LOCAL scope with a scope clause.

### 4.2.7. Erase Name Statement†

The statement

```
<<    erase <ADAMS_name> from <scope>       >>
```

will erase the class, codomain, or element name from the name space. Only the user who initially entered the name has erasure permission. Erasure of the name from the name space will not necessarily delete the class, codomain, or element that it denotes, unless it is referenced nowhere else in the ADAMS system. Readily, erasure of a name from the name space can cause problems if there remains executable programs that reference it. See the delete statement in Section 4.2.9, or warmstart in Section 6.2, for more details regarding element deletion and name space erasure respectively.

### 4.2.8. Exit_loop Statement

If the statement

```
<<    exit_loop   >>
```

is executed, the innermost *for_each* loop containing the statement is exited, in much the same way that `break;` will terminate an iteration statement in C, or C++. The <*loop_variable*> will denote the same element it denoted before executing the loop exit. On normal loop termination it is undefined.

There is no analog to the C `continue;` statement in ADAMS.

### 4.2.9. Delete Element Statement†

From time to time data elements must be deleted from the persistent data space altogether. (Names are *erased* from the name space; elements are *deleted* from the data space.) But, there must be constraints. Readily, an element that is the element of a set may not be deleted so long as it is a member of the set. Similarly, an element that is named in the name space may not be deleted so long as its name can still be referenced. Consequently, the statement

```
<<    delete <elem_desig>  >>
```

may have no effect whatever. In fact, it is questionable whether the statement even belongs in ADAMS at all!

To control unwanted element deletion, an invisible *ref_cnt* attribute is associated with every element which records the number of references to the element within either the ADAMS

data space or name space. When an element is inserted into a set, or assigned as image of a map, its *ref_cnt* is incremented; when it is removed from a set, replaced as a map image, or its name (if any) erased from the name space, its *ref_cnt* is decremented. Only elements with *ref_cnt* = 0 may actually be deleted, and this may occur only after all currently executing processes (which may have an ADAMS_var reference to the element) have terminated. Consequently, actual deletion is performed by a daemon, which rechecks that the *ref_cnt* is still zero, sometime after the delete statement is executed. The invocation of the deletion daemon can as well be made by any statement which decrements the *ref_cnt* to zero, making an explicit delete command superfluous.

### 4.2.10. For_each Loop Statement

The statement

```
<<    for_each  <loop_var>  in  <set_desig>  do
            .
            . <ADAMS/host language statements>
            .
            >>
```

provides a mechanism for looping over each element of <*set_desig* >, denoting the current element by <*loop_var* >, which must have been declared to be an ADAMS variable, and performing the intervening sequence of ADAMS and/or host language statements. On termination of the loop (except by a *loop_exit* statement) the <*loop_var* > is undefined. Semantically, the <*set_desig* > over which the loop is run is that set denoted on entrance to the loop, even though the <*set_desig* > may be modified within the loop body by element insertion or deletion.

For_each loops may be arbitrarily nested, but each must have a distinct <*loop_var* >. Attempt to assign to <*loop_var* > within the body of the loop will generate an error.

A small variant is created by use of the key word *for_all* instead of *for_each*, as in the following code that is frequently used to display elements and their assigned attribute values.

```
<<    for_each  x  in  graduate  do
<<          for_all  attr  in  view  do
<<                | x_value  char* |  <-  x.attr      >>
                  printf ("%s ", x_value);
                  >>
            printf ("\n");
            >>
```

Semantically, the use of *for_each* and *for_all* are identical. However, the use of *for_all* within the inner loop indicates to the preprocessor that this loop may be unwound to exploit multiple stream parallelism in environments that support it.

### 4.2.11. Insert Statement

The statement

```
<<    insert  <elem_desig>  into  <set_desig>  >>
```

inserts the designated element into the set. The class of <*elem_desig* > must be conformable to the *cose* of <*set_desig* >. If <*elem_desig* > is already a member of the set, the statement is interpreted as a *no_op*, not a statement failure.

### 4.2.12. Lock and Unlock Statements†

Because ADAMS is normally executed in a parallel processing environment, the use of locks is discouraged. However, for some applications they are essential. The statement

```
<<    lock  <elem_desig>          >>
```

grants the executing process an exclusive lock on <*elem_desig*> until either it is explicitly unlocked with the statement

```
<<    unlock  <elem_desig>        >>
```

or the process terminates. If the <*elem_desig*> denotes a set, then every element of the set is locked. Currently, the ADAMS syntax provides no mechanism for locking the set itself, as opposed to locking each of its constituent elements.

### 4.2.13. Open_ADAMS, Close_ADAMS Statements

The ADAMS name space, data space, and runtime system are attached to an applications program by

```
<<    open_ADAMS  job_id      >>
```

It must precede the first executable ADAMS statement in any program. Both ADAMS_var and ADAMS prototype statements are non-executable, and it is common to place these among other variable and procedure declarations ahead of the open_ADAMS statement. The last executed statement must be

```
<<    close_ADAMS  job_id     >>
```

which shuts down ADAMS and flushes any cached elements of the data space and name space to persistent storage.

The entire ADAMS execution, from *open_ADAMS* to *close_ADAMS* is treated as a default transaction, whose transaction identification is denoted by the string variable *job_id*.†[17]

### 4.2.14. Procedure Header Statement

Host language procedures containing ADAMS statements may be separately compiled. Such procedures need not contain an open_ADAMS or a close_ADAMS statement, unless they will be the first, or last, executed ADAMS statements, and opening and/or closing ADAMS is not handled by the main program. However, because the ADAMS name space and the program's name space are disjoint, ADAMS names and ADAMS expressions may not be passed as parameters. To pass an ADAMS designator to a subprocess, one must employ an **ADAMS procedure** which has a different kind of procedure header, procedure end, and procedure invocation format.

An **ADAMS_procedure** statement has the form:

```
<<    ADAMS_procedure <proc_name> ( <formal_param_list_ADAMS>
                                    $ <formal_param_list_host> $ ) >>
```

Formal ADAMS parameters must be ADAMS variables. Each entry in the <$formal\_param\_list_{ADAMS}$> is a pair consisting of <*class_name*> and <*ADAMS_var*>, where the class name declares the class of element that will be passed as actual parameter.†[18] A runtime check is conducted to ensure that the <*elem_desig*> passed as an actual parameter is conformable to this declared class. If host language parameters are also being passed to the procedure, they must be coded in the fashion expected by the host language and delimited by $ ... $.

---

†[17] Transaction management is not currently supported, however a variable name must be provided to parse correctly. There may be several *open_ADAMS* or *close_ADAMS* statements in the source code, but the system must first be closed before issuing another *open_ADAMS* statement.

†[18] Currently, each <*ADAMS_var*> appearing in a <$formal\_param\_list_{ADAMS}$> must be explicitly declared in an ADAMS variable statement. It is anticipated that in future versions, inclusion in this list will automatically generate the required ADAMS variable declaration code.

An ADAMS procedure must be terminated by the **procedure end** statement

```
<<    end_ADAMS_procedure     >>
```

As an example of a complete ADAMS procedure, consider the following utility code which displays all the attributes associated with a set of elements in a relational style.

```
<<    ADAMS_var  element_set           >>    /* Formal parameter */

<<    ADAMS_procedure  show_set  ( GENERIC_SET  element_set
                                 $ FILE *outfile $ )         >>
      /*
      **  Display ALL the attributes associated with the
      **  elements of 'element_set' on in 15-character fields
      **  as a table (relational style) on 'outfile'.
      */
      int        set_size, attr_set_size;
      char       class_name[21], value_str[30], separator[256];
<<    ADAMS_var  attr_set, attr, x                   >>

<<    | set_size int |  <-  element_set.cardinality  >>
      if (set_size == 0)
            {
            fprintf      (outfile, "\tEMPTY SET\n");
            goto  exit_proc;           /* NOTE: can't 'return'  */
                                       /* MUST exit through end */
            }
                                       /* First, get CLASS of a */
                                       /* typical set element   */
<<    | class_name char* 20 | <- element_set.element_of.class_of  >>
      if (strcmp(class_name, "") == 0)
            {
            fprintf      (outfile, "\tCan't determine CLASS of elements\n");
            goto  exit_proc;
            }
                                 /* Now, get all attributes       */
<<    attr_set  <-  var class_name . attributes_of         >>
<<    | attr_set_size int |  <-  attr_set.cardinality      >>
      if (attr_set_size == 0)
            {
            fprintf      (outfile,
                  "\tNo attributes defined for elements of CLASS '%s'\n",
                                 class_name);
            goto  exit_proc;
            }
      strcpy (separator, "");
<<    for_each  attr  in  attr_set   do
<<          | value_str char* 15 | <- attr.name_of  >>
            fprintf (outfile, "| %-15.15s ", value_str);
            strcat (separator, "+-----------------");
            >>
      fprintf (outfile, "|\n");
      strcat (separator, "+");
      fprintf (outfile, "%s\n", separator);

<<    for_each  x  in  element_set  do
<<          for_all  attr  in attr_set  do
<<                | value_str char* 15 | <- x.attr  >>
                  fprintf (outfile, "| %-15.15s ", value_str);
                  >>
            fprintf (outfile, "|\n");
            >>
```

```
        fprintf (outfile, "%s\n", separator);
        fprintf (outfile, "\n");
exit_proc:
<<      end_ADAMS_procedure >>
```

Show_set Procedure
Figure 4-1.

There are several interesting features of note in this procedure. First, because the class of set to be displayed is unknown, the predefined class GENERIC_SET is specified for *element_set*. Second, several predefined attributes and maps must be used to determine the actual class of *element_set*, so that we can retrieve its set of attributes. Third, one must exit through the end_ADAMS_procedure statement; one cannot use a host language `return;` statement in the usual fashion.[19] Fourth, a host language file pointer *outfile* has been passed to direct the output to the correct file. Finally, the usual { ... } delimiters surrounding the body of the procedure are missing; such delimiters if needed to compile the host code are automatically provided by the ADAMS procedure header and end_ADAMS_procedure statements.

There are no ADAMS functions, only ADAMS procedures; and no ADAMS procedure may return a value except by making an assignment to a formal ADAMS_var parameter, in which case the effect is identical to making the assignment to the corresponding actual parameter. Also, because ADAMS variables are currently global to the file of source code in which they occur, one **cannot** write recursive ADAMS procedures. This could be rectified with a better preprocessor.

### 4.2.15. Procedure Invocation Statement

ADAMS procedures are invoked by a statement of the form:

`<<` `invoke` <*proc_name*> `(` <$actual\_param\_list_{ADAMS}$> `$` <$actual\_param\_list_{host}$> `$ )` `>>`

where <$actual\_param\_list_{ADAMS}$> is a list of <*elem_desig*> which may optionally be comma delimited, and <$actual\_param\_list_{host}$> conforms to the expected usage of the host language.

For example, the statement

```
<<    invoke  show_set  ( undergrad $ stdout $ )      >>
```

will display all student elements in the set *undergrad*, whereas

```
<<    invoke  show_set  ( { x in graduate | x.gpa >= 3.0 }
                          $ stdout $ )                >>
```

will display those *graduate* students with *gpa* $\geq$ 3.0.

### 4.2.16. Procedure Prototype Statement

A prototype of each ADAMS procedure must be provided in any source code which invokes the procedure.[20] The **ADAMS prototype** statement has the form:

`<<` `ADAMS_prototype` <*proc_name*> `(` <$class\_list_{ADAMS}$> `$` <$type\_list_{host}$> `$ )` `>>`

where the <$class\_list_{ADAMS}$> consists of just the <*class_name*>'s used in the formal parameter

[19] We will probably include an ADAMS_return statement in the next version to eliminate this awkward construction.

[20] Note that, with our current preprocessing approach, one cannot create an include file of all relevant ADAMS prototype statements, because the *adams* preprocessor does not expand the `# include` directive; *cci* does.

list declaration, and <*type_list*$_{host}$> conforms to the expectations of the host language. The ADAMS prototype statement is non-executable, and may occur anywhere preceding the first invocation of the procedure.

For example, any code invoking the *show_set* procedure must include the prototype

```
<<    ADAMS_prototype  show_set ( GENERIC_SET $ FILE* $ )  >>
```

### 4.2.17. Remove Statement

An element may be removed from a set by the statement

```
<<    remove  <elem_desig>  from  <set_desig>  >>
```

If <*elem_desig*> is not an element of the set, the statement is treated as a *no_op*, not as a statement failure.

### 4.2.18. Rescope Statement†

ADAMS names can be moved from one name space hierarchy to another with the statement

```
<<    rescope  <ADAMS_name>  to  <scope>   >>
```

where <*scope*> may be either SYSTEM, TASK, USER, or LOCAL. Changing name visibility in this way can have unforeseen consequences. See the discussion in Section 6.1, or in [PFW88].

### 4.2.19. Start, End Transaction Statements†

The entire sequence of executable ADAMS statements, beginning with the open_ADAMS statement and terminating with a close_ADAMS statement is a transaction. That is, either all statements succeed or all are considered to fail. In the event of failure, all statements must be re-executed. If transactions are nested, as in [Mos85], then only the failed transactions need be re-executed. A nested transaction is initiated by the statement

```
<<    tr_start <trans_id>  >>
```

where <*trans_id*> is any unique character string denoting this transaction. A transaction is terminated with

```
<<    tr_end  <trans_id>   >>
```

## 4.3. Predefined ADAMS Classes, Codomains, Attributes, and Maps

For convenience, ADAMS provides 11 predefined classes, 3 predefined codomains, and 7 predefined attributes and maps. Because of their ubiquity in database applications, the three codomains are STRING, INTEGER, and REAL. The first is an *alphanumeric* codomain, while the latter two are *numeric*. The undefined constants, or *udf*, are "", "-999999999", and "-999999999.0" respectively.

Three of the predefined classes are the ATTRIBUTE classes STRING_ATTR, INTEGER_ATTR, and REAL_ATTR with STRING, INTEGER, and REAL codomains, respectively. Four more classes are the generic classes GENERIC_ELEMENT, GENERIC_SET, GENERIC_ATTR, and GENERIC_MAP, which are sometimes useful when defining formal parameters of ADAMS procedures whose actual parameters are not known. (These are equivalent to CLASS, SET, ATTRIBUTE, and MAP respectively, which being reserved words cannot be used as class names.) Generic attribute and map sets are defined by ATTR_SET and MAP_SET, and maps with these images are defined by ATTR_SET_MAP and MAP_SET_MAP.

Six predefined attributes are *cardinality*, *name_of*, *class_of*, *element_class_of*, *image_of* and *codomain_of*. The first is an INTEGER_ATTR, which is used to test the

cardinality of ADAMS sets. One would very much like to have Boolean functions of the form *empty*(<*set_desig* >) and *singleton*(<*set_desig* >), however the current preprocessing paradigm prohibits this. ADAMS procedures cannot return a host language value, and ADAMS names (or expressions) cannot be passed to a host language procedure. An integrated compiler could rectify this. The latter five attributes are STRING_ATTR's; they return the names of elements (*name_of* any element, which returns "" if the element is unnamed), of classes (*class_of* any element, *element_class_of* any set, and *image_of* of any map), or of codomains (*codomain_of* any attribute). Since all classes and codomains must be named, these attributes are always well defined. The class, or codomain name, can then be used as <*var_name* >'s (Section 4.1.1) to designate the corresponding class, or codomain. Note that there are no general <*class_desig* > or <*codomain_desig* > constructs, so these can only be designated by name.

Four predefined maps are *element_of* , *random*†[21], *attributes_of* , and *maps_of* . The first two, given a non-empty <*set_desig* > as argument, return an element from the set. The latter two return the set of all attributes, or maps, associated with a <*class_name* > and its superclasses as an ATTR_SET and MAP_SET respectively.

---

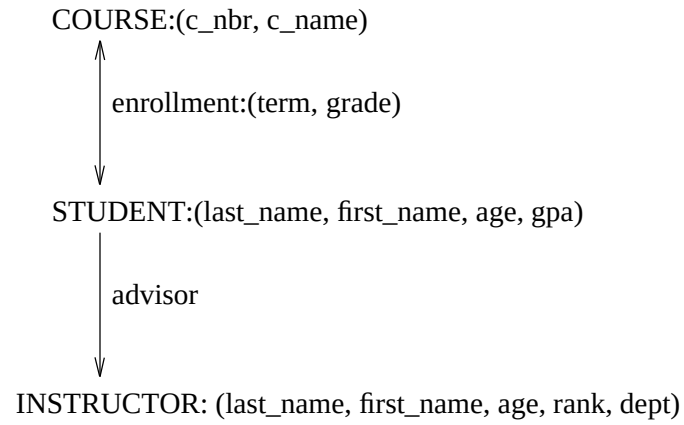
†[21] Both the *element_of* and *random* maps return as their image an arbitrary element of their set arguments. The difference is that repeated invocations of *element_of* will always return the same element, whereas *random* will return some random sequence of elements. The map *random* has not yet been implemented in the current version.

## 5. More ADAMS Examples

In this section we examine several complete programs, procedures, and code fragments which make use of the more advanced ADAMS features described in the preceding Section 4. These include recursive class declaration, ADAMS procedures, subscripted element identifiers, and dynamic class modification. Not all of the code is good, in the sense of representing the best use of ADAMS. But, it will illustrate the flexibility of the language, together with many of its capabilities. Again, all code given here has been compiled and executed.

### 5.1. Graphs and Relationships

General many-to-many relationships are fundamental in database design, c.f. [Che76]. For example, if we wished to extend our school database example to include classes offered by the university, together with student enrollment in these courses, the E-R diagram might look like:




One way of representing such relationships in ADAMS is by declaring a COURSE class, together with a COURSE_MAP class. Then an ENROLLMENT class can be declared using two maps

```
<<    course   instantiates_a  COURSE_MAP       >>
<<    student  instantiates_a  STUDENT_MAP      >>
<<    ENROLLMENT  isa  CLASS
            having  attrs = { term, grade }
            having  maps  = { student, course } >>
```

This approach simply represents a binary, many-to-many relationship as a set of ordered pairs of many-to-one functional maps.

Mathematically, a directed graph is simply a binary relationship on the set of vertices comprising the graph. Each ordered pair of the relationship is called an edge. We can use the same approach to define a class of directed graphs in ADAMS.

```
main ()
      /*
      **  This program defines a generic space of
      **  of directed graphs, whose vertices and
      **  edges may be labelled.
      **
      **  As defined, the labels on both the VERTEX
      **  and EDGE elements are string labels, but
      **  either class declaration can be modified
      **  to provide other kinds of labels.
      **
      **  A GRAPH is regarded as a SET of VERTEX elements
      **  and a SET of EDGE elements.
```

```
      */
      {
<<    open_ADAMS  job_id  >>

<<    label  instantiates_a  STRING_ATTR      >>

<<    VERTEX isa  CLASS
            having  attrs = { label }  >>
<<    VERTEX_MAP  isa  MAP
            with image VERTEX          >>
<<    v1     instantiates_a  VERTEX_MAP >>
<<    v2     instantiates_a  VERTEX_MAP >>
<<    EDGE   isa  CLASS
            having  attrs = { label }
            having  maps  = { v1, v2 } >>

<<    VERTEX_SET  isa  SET
            of  VERTEX  elements       >>
<<    EDGE_SET    isa  SET
            of  EDGE  elements         >>
<<    VERTEX_SET_MAP  isa  MAP
            with image VERTEX_SET      >>
<<    EDGE_SET_MAP    isa  MAP
            with image EDGE_SET        >>
<<    vertices instantiates_a  VERTEX_SET_MAP >>
<<    edges    instantiates_a  EDGE_SET_MAP   >>

<<    DI_GRAPH  isa  CLASS
            having  attrs = { label }
            having  maps  = { vertices, edges }     >>

<<    close_ADAMS  job_id >>
      }
```

A Definition of Directed Graphs
Figure 5-1.

While the definition of directed graphs given above is a very serviceable one, an alternative, slightly less general definition will further our exploration of the ADAMS language and its features. If we will never associate data with individual edges, e.g. edge labels, then there is no need to explicitly create an EDGE class. Instead, we can simply designate the set of vertices, $\{ y \}$, for which there is an edge $(x, y)$ as the **neighbors** of $x$. Similarly, we will call the set of all vertices, $\{ z \}$, which are reachable by a directed path from $x$, its **closure**. This graph terminology is extensively developed in [Pfa77]. The problem is that *nbhrs* is a map defined on the class of VERTEX elements into the class of sets of VERTEX elements. It is a recursive definition. To effect this we use a FORWARD class declaration as in the following code:

```
main ()
      /*
      **  This program defines a generic space of
      **  of directed graphs
      **
      **  A GRAPH is regarded as a SET of VERTEX elements
      **  together with a map 'nbhrs'.
      */
      {
<<    open_ADAMS  job_id  >>

<<    id  instantiates_a  STRING_ATTR        >>
```

```
<<    VERTEX isa  CLASS,  FORWARD              >>

<<    VERTEX_MAP  isa  MAP
            with image VERTEX          >>
<<    VERTEX_SET  isa  SET
            of  VERTEX  elements       >>
<<    VERTEX_SET_MAP  isa  MAP
            with image VERTEX_SET            >>
<<    nbhrs  instantiates_a  VERTEX_SET_MAP   >>
<<    vertices instantiates_a  VERTEX_SET_MAP >>

<<    VERTEX isa  CLASS
            having  attrs = { id }
            having  maps  = { nbhrs }  >>

<<    DI_GRAPH  isa  CLASS
            having  attrs = { id }
            having  maps  = { vertices }     >>

<<    close_ADAMS  job_id >>
      }
```

Alternative Definition of Directed Graphs
Figure 5-2.

To access all vertices that are reachable from from a designated *vertex* one could write the following *closure* procedure. We observe that this is simply a generalization of the parts explosion problem, where one considers VERTEX elements to be a PART elements, and the map *nbhrs* to be a *is_a_subpart_of* map. Then the *closure* is the set of all parts of which the designated PART is *contained_in* .

```
<<    ADAMS_var   vertex, reachable    >>    /* formal parameter */

<<    ADAMS_procedure  closure ( VERTEX  vertex, VERTEX_SET reachable )  >>
      /*
      **  Determine the closure of 'vertex',
      **  that is, the set of all vertices that are reachable from it,
      **  and return this set as 'reachable'.
      */
      int        bndy_size;
<<    ADAMS_var  v1, v2, old_bndy, new_bndy   >>

                        /* initialize the set ADAMS_vars */
<<    old_bndy   instantiates_a  VERTEX_SET
            scope is LOCAL                   >>
<<    new_bndy   instantiates_a  VERTEX_SET
            scope is LOCAL                   >>

<<    insert  vertex  into  old_bndy         >>
      bndy_size = 1;
      while (bndy_size > 0)
            {
<<          new_bndy  <-  NULLSET     >>
<<          for_each  v1  in  old_bndy  do
<<                for_each  v2  in  v1.nbhrs  do
<<                      insert  v2  into  reachable      >>
<<                      insert  v2  into  new_bndy       >>
                        >>
                  >>
<<          old_bndy  <-  new_bndy    >>
```

```
<<          | bndy_size int |  <-  old_bndy.cardinality >>
            }

<<     end_ADAMS_procedure >>
```

A Graph Closure Procedure
Figure 5-3.

We hasten to observe that the primary purpose of ADAMS is to designate and access portions of a persistent data space, not to perform computational operations within the data space itself as this procedure does. Nevertheless, this procedure does illustrate that the ADAMS language does possess rather powerful designational capabilities.

## 5.2. Arrays and Matrices

ADAMS does not explicitly provide either an array or matrix class, however the ability to subscript element names allows the user to do so in a variety of ways. Let us consider the definition of doubly subscripted, $m \times n$ real matrices. The most obvious, but not necessarily best, way to regard a matrix as a doubly subscripted set of elements, each of which has single associated real attribute, say *r_val*, as in:

```
<<    val  instantiates_a  REAL_ATTR      >>
<<    REAL_MATRIX_ELEMENT isa  CLASS
            having  attrs = { r_val }     >>
<<    x[*,*] instantiates_a  REAL_MATRIX_ELEMENT      >>
```

The value of any matrix element can now be denoted by

$$x[i,j].r\_val.$$

But, this is a rather unsatisfactory approach for several reasons. The primary disadvantage is that the matrix is not a single ADAMS element, but rather a family of subscripted elements. One cannot easily pass the matrix as a parameter to procedures.

The approach we actually use in ADAMS is based on subscripting attributes, not the elements on which the attribute is defined [PfF90]. For example, consider the class declaration

```
main()
      /*
      **  Test subscripted definitions and matrix operations
      */
      {
<<    open_ADAMS  job_id  >>

<<    REAL_ATTR  isa  CODOMAIN
            consisting of #[-|+]?[0-9]*.[0-9]*#
            treat_as numeric
            udf = '0.0'                >>

<<    r_val[*,*] instantiates_a REAL_ATTR     >>

<<    3x3_REAL_MATRIX isa CLASS
            having { r_val[1..3,1..3] }      >>
<<    4x4_REAL_MATRIX isa CLASS
            having { r_val[1..4,1..4] }      >>
<<    3x4_REAL_MATRIX isa CLASS
            having { r_val[1..3,1..4] }      >>

<<    close_ADAMS  job_id >>
      }
```

Declaration of Real Matrix Classes
Figure 5-4.

which we will test below with a matrix multiplication. There are three features of interest here. First, it is seen that a matrix is a single ADAMS element with multiple, subscripted attributes defined on it — *r_val*[1,1], *r_val*[1,2], ..., *r_val*[3,2], *r_val*[3,3], *r_val*[3,4]. Second, we use <*subscript_range*>'s to compactly describe these enumerated sets. Finally, in the case of sparse matrices in which many matrix elements are zero, we do not want to specifically store a zero value; we would prefer to simply regard the element as undefined. This is how ADAMS handles sparse matrices. Only non-zero attributes are represented. But, this requires the re-declaration of the REAL_ATTR codomain so that the *udf* value is '0.0', instead of the system default
For this user, REAL_ATTR will be as declared here, instead of the predefined REAL_ATTR codomain provided by the ADAMS system. (C.f. Section 6 for a fuller explanation of this.)

Because the following code is only a test program, it instantiates the matrices $x$, $y$, and $y$, with LOCAL scope; we do not want them to be persistent. It then initializes them with arbitrary values. Values below the main diagonal of $y$ are undefined, or zero. The two matrices are then displayed, as initialized, multiplied, and the result $z$ displayed.

```
main()
      /*
      **  Test matrix multiply.
      */
      {
      char   jobid[10];
      int    i, j, k;
      float  x_value, y_value, z_value, sum;

<<    open_ADAMS  job_id  >>

<<    x      instantiates_a 3x3_REAL_MATRIX    scope is LOCAL      >>
<<    y      instantiates_a 3x4_REAL_MATRIX    scope is LOCAL      >>
<<    z      instantiates_a 3x4_REAL_MATRIX    scope is LOCAL      >>

                                /* initialize matrix 'x' */
      for (i=1; i<=3; i++)
            {
            printf ("");
            for (j=1; j<=3; j++)
                  {
                  x_value = -5 + 2*i + 3*(i-j);
                  printf ("%6.1f  ", x_value);
<<                x.r_val[i,j] <- | x_value float | >>
                  }
            printf ("0);
            }
                                /* initialize matrix 'y' */
                                /* as upper diagonal     */
      for (i=1; i<=3; i++)
            {
            printf ("");
            for (j=i; j<=4; j++)
                  {
                  y_value = 4 + 3*i - 2*(j-i);
                  printf ("%6.1f  ", y_value);
<<                y.r_val[i,j] <- | y_value float | >>
                  }
            printf ("0);
            }

                                /* display 'x' and 'y' */
      printf ("x =0);
      for (i=1; i<=3; i++)
```

```
                {
                printf ("");
                for (j=1; j<=3; j++)
                        {
<<                      | x_value  float | <- x.r_val[i,j]      >>
                        printf ("%6.1f  ", x_value);
                        }
                printf ("0);
                }
        printf ("y =0);
        for (i=1; i<=3; i++)
                {
                printf ("");
                for (j=1; j<=4; j++)
                        {
<<                      | y_value  float | <- y.r_val[i,j]      >>
                        printf ("%6.1f  ", y_value);
                        }
                printf ("0);
                }
                                        /* 'z' <- 'x' * 'y' */
        for (i=1; i<=3; i++)
                {
                for (j=1; j<=4; j++)
                        {
                        sum = 0;
                        for (k=1; k<=3; k++)
                                {
<<                              | x_value  float | <- x.r_val[i,k] >>
<<                              | y_value  float | <- y.r_val[k,j] >>
                                sum += x_value * y_value;
                                }
<<                      z.r_val[i,j] <- | sum  float |          >>
                        }
                }
                                        /* display 'z' */
        printf ("z =0);
        for (i=1; i<=3; i++)
                {
                printf ("");
                for (j=1; j<=4; j++)
                        {
                <<      | z_value  float | <- z.r_val[i,j]      >>
                        printf ("%6.1f  ", z_value);
                        }
                printf ("0);
                }

<<      close_ADAMS  job_id >>
        }
```

A Matrix Multiply Program to Test ADAMS Subscripting
Figure 5-5.

which generates as output

```
x =
          -3.0    -6.0    -9.0
           2.0    -1.0    -4.0
           7.0     4.0     1.0
y =
           7.0     5.0     3.0     1.0
           0.0    10.0     8.0     6.0
           0.0     0.0    13.0    11.0
z =
         -21.0   -75.0  -174.0  -138.0
          14.0     0.0   -54.0   -48.0
          49.0    75.0    66.0    42.0
```

Again, we would emphasize that ADAMS is not intended to be a computational language; it should not be used for matrix operations. But, this example does illustrate some of the more interesting possibilities that subscripted identifiers provides.

It also illustrates a major deficiency in the language as it now exists!

One would like to be able to define and implement generic matrix operators whose dimensions *m* and *n* are variable. One would like to have parameterized class declarations of the form

```
<<    $1_x_$2_REAL_MATRIX isa CLASS
            having { r_val[1..$1,1..$2] } >>
```

where a macro substitution of the parameters $1 and $2, say by 3 and 4, as in the class name 3_*x*_4_*REAL_MATRIX* denotes the corresponding element class. In the formal syntax of Section 7, we find that class names can contain **parameter segments**, or <*param_seg* >, which constitute stubs for developing this capacity in the next version, but for now they are not implemented.

## 5.3. Dynamic Schema Modification

Because attributes are ADAMS elements, because elements may be added and deleted from sets, and because the attributes associated with class declarations are sets, one may dynamically alter class declarations in ADAMS. This is sometimes called *dynamic schema evolution*. A sample program that we use to modify existing class declarations in large applications, without recompiling and reloading the entire database is:

```
main()
      /*
      **  This program adds new attributes to CLASS->attrs
      */
      {
      char   class_name[80], attr_name[80];

<<    open_ADAMS  job_id  >>
<<    ADAMS_var   a, a_set      >>

      printf ("enter CLASS to be modified (# to quit) > ");
      scanf  ("%s", class_name);
      while (*class_name != '#')
            {
<<          a_set <- var class_name->attrs                >>
            printf ("\tcurrent attributes in %s->attrs\n{ ",
                                    class_name);
<<          for_each a in a_set do
<<                | attr_name char* | <- a.name_of        >>
                  printf (" %s", attr_name);
                  >>
            printf (" }\n");

            printf ("enter attribute to be added to class '%s' > ",
```

```
                        class_name);
                scanf  ("%s", attr_name);
<<              insert var attr_name into var class_name->attrs>>

                printf ("\tcurrent view of associated %s attributes\n\t{ ",
                                        class_name);

<<              a_set <- var class_name->attrs                  >>
<<              for_each a in a_set do
<<                    | attr_name char* | <- a.name_of         >>
                      printf (" %s", attr_name);
                      >>
                printf (" }\n");

                printf ("\tcurrent view of ALL %s attributes\n",
                                        class_name);
                printf ("\t\tincluding superclass attributes\n\t{ ");
<<              for_each a in  var class_name.attributes_of  do
<<                    | attr_name char* | <- a.name_of         >>
                      printf (" %s", attr_name);
                      >>
                printf (" }\n");

                printf ("enter CLASS to be modified (# to quit) > ");
                scanf  ("%s", class_name);
                }

<<      close_ADAMS  job_id                 >>
        }
```

Dynamic Addition of Attributes to a Class Declaration
Figure 5-6.

Readily, similar code can be written to delete attributes, or to add and delete maps from a declaration.

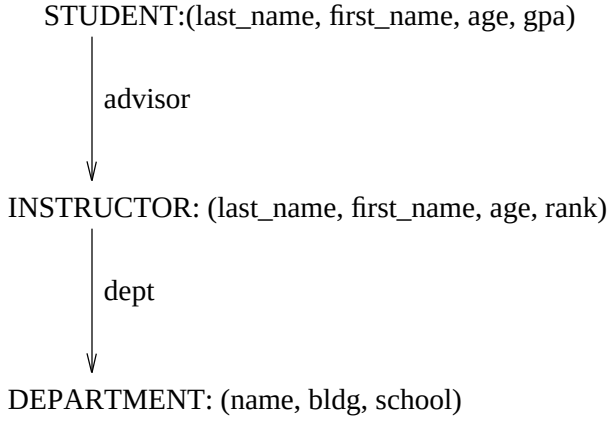
Addition of new attributes or maps cannot invalidate existing application code; but deletion can cause problems if the attribute/map is explicitly named. Care must be taken.

## 5.4. Relational Joins

The most important relational joins are lossless joins. As discussed in Section 2.5, a join $r \bowtie s$ is lossless if the join attribute(s) constitute a key to one of operand relation sets, say of $s$; that is if the join attribute(s) function as a symbolic pointer to a unique element in $s$, or *foreign key*. In ADAMS, such joins are represented, not by symbolic attribute value(s), but rather by a map function declared on the class $R$ of tuples in $r$ into the class $S$ of tuples in $s$. The *advisor* map, defined in Figure 3-1, is representative of such a many-to-one relationship from STUDENT elements to INSTRUCTOR elements, which in the relational model would typically be implemented by an equi-join of an *advisor* attribute in STUDENT tuples with the *last_name* attribute in INSTRUCTOR tuples. In ADAMS, the corresponding INSTRUCTOR element is obtained by the <*elem_desig* >

```
t.advisor
```

An important advantage of representing such symbolic pointers by maps is that maps can be concatenated in designational expressions, thereby eliminating the need for multiple joins. For example, if in the development of the school database we had created a DEPARTMENT class to represent individual department elements at the university, we could replace the *dept* attribute in the INSTRUCTOR class with a *dept* map. Now the E-R diagram would look like




and we could obtain a student's major department, that is the department of his, or her, advisor by the expression

```
t.advisor.dept
```

and the name of that department by the expression

```
t.advisor.dept.name
```

All CS students could be denoted by

```
{ t in students | t.advisor.dept.name = 'CS' }
```

No joins have been required.

Moreover, to change the structure of the database as we have just done does not require a reformating of the INSTRUCTOR elements. It is sufficient to simply remove the *dept* attribute from the set of attributes associated with the INSTRUCTOR class and insert a *dept* map into the associated set of maps, as described in Section 5.3 above.

While virtually all of the join operations required in the relational model can be replaced by map constructs, there may occur cases where one wants to execute a natural join given a relational model. The following code, although complex, does so with considerable efficiency when there is only one join attribute. Its primary purpose is to illustrate that it is possible to completely emulate the relational algebra in ADAMS, and thus ADAMS has at least the power of a relational database.

```
<<     ADAMS_var  r, s, join      >>     /* Formal parameters */

<<     ADAMS_procedure  nat_join  ( GENERIC_SET r, GENERIC_SET s,
                                    GENERIC_SET join )           >>
       /*
       **  This procedure implements the natural join of 'r' with 's',
       **  (i.e. over all common attributes) to form 'join'
       **  Note that both 'join' and its associated classes are
       **  LOCAL, i.e. non-persistent.
       */
       char       class_name[81], r_value[81], s_value[81];
       char       JOIN_TUPLE[21], JOIN_SET[21];
       char       generate_name();
       int        tuples_join;
<<     ADAMS_var  r_tuple, s_tuple, s_tuples, R, S, RS      >>
<<     ADAMS_var  a, join_attrs, t      >>

                                  /* First determine the schema */
                                  /* R and S  of  'r' and 's'   */
<<     | class_name char* 80 |  <-  r.element_of.class_of    >>
```

```
<<     R  <-  var class_name . attributes_of                >>
<<     | class_name char* 80 |  <-  s.element_of.class_of   >>
<<     S  <-  var class_name . attributes_of                >>

<<     join_attrs  <-  R intersect S                        >>
<<     | size int |  <-  join_attrs.cardinality            >>
       if (size == 0)
             {
             printf ("\tthe join operands have no attributes in common\n");
             printf ("\treturning a NULLSET\n");
<<           join  <-  NULLSET   >>
             goto  exit_proc;
             }
                              /* Now, declare the join_set  */
                              /* and join_tuple classes     */
                              /* NOTE! We must generate new */
                              /* names for every invocation */
       strcpy (JOIN_TUPLE, generate_name());
       strcpy (JOIN_SET,   generate_name());
<<     RS  <-  R  union  S                          >>
<<     var JOIN_TUPLE  isa  CLASS
             having  RS
             scope is LOCAL                          >>
<<     var JOIN_SET    isa  SET
             of  var JOIN_TUPLE  elements
             scope is LOCAL                          >>

       if (size == 1)
             {             /* Better, indexed retrieval join */
                           /* 'a' is the unique join attr    */
<<           a <- join_sttrs.element_of >>
<<           for_each r_tuple in r  do
<<                 s_tuples <- { s_tuple in s | s_tuple.a = r_tuple.a } >>
<<                 for_each  s_tuple  in  s_tuples  do
                                 /* concatenate with r_tuple */
<<                       t  instantiates_a  JOIN_TUPLE,
                                       scope is LOCAL      >>
<<                       for_each  a  in  R  do
<<                             t.a  <-  r_tuple.a        >>
                               >>
<<                       for_each  a  in  S  do
<<                             t.a  <-  s_tuple.a        >>
                               >>
<<                       insert  t  into  join           >>
                         >>
                 >>
             }
       else
             {             /* Brute force, iteration join */
<<           for_each  r_tuple  in  r  do
<<                 for_each  s_tuple  in  s  do
                         tuples_join = 1;
<<                       for_each  a  in  join_attrs  do
<<                             | r_value  char*  80 |  <-  r.a  >>
<<                             | s_value  char*  80 |  <-  s.a  >>
                               if (strcmp(r_value, s_value) != 0)
                                     {
                                     tuples_join = 0;
<<                                   exit_loop    >>
                                     }
                               >>
                         if (tuples_join)
                               {     /* concatenate with r_tuple */
```

```
<<                                  t  instantiates_a  JOIN_TUPLE,
                                           scope is LOCAL      >>
<<                                  for_each  a  in  R  do
<<                                         t.a  <-  r_tuple.a  >>
                                           >>
<<                                  for_each  a  in  S  do
<<                                         t.a  <-  s_tuple.a  >>
                                           >>
<<                                  insert  t  into  join      >>
                                    }
                              >>
                        >>
               }
exit_proc:
<<     end_ADAMS_procedure >>

char   *generate_name ()
       /*
       **  Generate a unique name on each iteration
       */
       {
       static char  name[5] = "abcd";
       static char  prefix  = 'a';

       *name = prefix++;
       return  name ;
       }
```

A General Relational Join Procedure
Figure 5-7.

Of greatest interest in this code is the need to declare the JOIN_TUPLE and JOIN_SET classes on the fly. But, classes must be named; and the name must be distinct for every invocation. Consequently the code generates synthetic names and uses the *var <host_variable>* construct described in Section 4.1.1. Equi-joins over different attributes, or more general theta-joins, could be similarly coded, by passing the appropriate join attributes as formal parameters.

## 6. The ADAMS Name Space

Although we have repeatedly been using class and element names in our code and examples, and have mentioned the ADAMS name space, we have really said very little about it. And yet, it is at the heart of the ADAMS language — it is what makes ADAMS more than just another database system. In this section, we examine the nature of the ADAMS **name space**.

It should be evident that, at a fundamental level, all access to data requires the existence of symbolic names that denote data elements. The most familiar example is the use of symbolic variable names to denote the data values used in a host language process or procedure. Of course, not all data elements need to be so named. In many languages, storage can be dynamically allocated to create linked structures of considerable complexity in which constituent elements may be unnamed. Use of the ADAMS statement

```
<<    <ADAMS_var>  instantiates_a  <class_name>    >>
```

is in many ways analogous to the use of

```
<pointer_var> = (<type> *) malloc (sizeof(<type>));
```

in C, or

```
new (<pointer_var>);
```

in Pascal. It instantiates an unnamed element of the appropriate class to which *<ADAMS_var >* points (or denotes). Nevertheless, every process requires at least one symbolic variable name, it might be *root* or *current_node* in the case of a tree structure, to gain access into the linked structure itself. In [KhC86], Khoshafian and Copeland make clear that there exist a large variety of other mechanisms for denoting data elements, that is for establishing element identity, without explicitly naming them. For example, individual elements (or tuples) in a set (or relation) may be denoted by iterating over the set or by retrieval on a key attribute. This is true; but the set itself must be named, or its identity eventually derived from a set expression involving symbolic names.

One can never eliminate symbolic naming. In many respects it is what make a programming system into a "language".

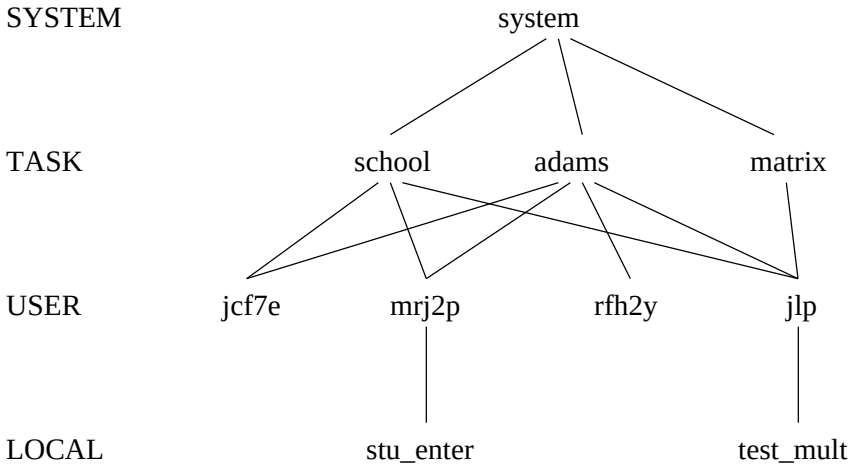
A cardinal rule of programming is that symbolic names should be mnemonic, not artificial. But this may be difficult to achieve. Humans have a relatively small set of meaningful words for most concepts. Frequently, the same mnemonic name is appropriate for different data elements. In scientific programming, *x, f*, and *i* are heavily overworked names for real numbers, functions, and subscripts respectively. However, every symbolic name must be uniquely bound to a single element that it denotes. In particular, the same persistent name cannot be arbitrarily reused in different programs. Consequently, our store of appropriate mnemonic names is quickly exhausted.

A technique that is commonly used to expand a name space and provide for name reusability is to partition it. The same symbolic name appearing in distinct subsets of the partition are assumed to denote different elements; they are, in fact, different names. In traditional programming languages, the variable *x* declared in two separate procedures denotes different variables. The name space is implicitly partitioned with respect to the procedure names, and, if it is a recursive language, with respect to different procedure invocations. Because of this, the same symbolic name, say *x*, may be used in two different programs, or in two different invocations of the same program, to denote different data elements. Partitioning the variable name space with respect to individual procedures is so convenient that we almost do it instinctively.

### 6.1. Name Space Hierarchy and Task Partitions

ADAMS provides for both an explicit partitioning of its name space to allow use of identical data names by different users, and for a hierarchical subdivision to facilitate sharing of other data names. Currently, the persistent ADAMS name space consists of three levels, called

visibility **scopes**. Names at SYSTEM scope are visible to all users and all processes. Names of USER scope are only visible to a single user. Both the user, and his name space, are identified by his login id. Consequently the data name *x* declared by user *mrj2p* is invisible to user *rfh2y*, and distinct from any name *x* declared by *rfh2y*. In between the USER and SYSTEM scopes is a TASK scope consisting of various named tasks. Names in a particular TASK name space may be visible to some users, but not necessarily all users. In addition, there is a non-persistent LOCAL scope consisting of those names which are visible only to the executing process. Schematically, a snapshot of the ADAMS name space might look like




Snapshot of the ADAMS name space
Figure 6-1

when users *mrj2p* and *jlp* are executing a student entry program and test of matrix multiply, respectively. Each of the users *jcf7e, mrj2p, rfh2y* and *jlp* has his own private USER name space. The data names, and hence corresponding data elements, in the subspaces of the TASK partition, *school* and *adams* are visible to both *jcf7e* and *mrj2p*; only those in *adams* are visible to *rfh2y*; while all TASK name space partitions a visible to *jlp*. The task partition *adams* is visible to all users, by default. An ADAMS statement embedded in a host language procedure may reference any symbolic name that is visible to that procedure.

When an ADAMS name, or <*actual_name*>, is encountered in an ADAMS statement, say in the process *test_mult*, it is first compared with the list of known ADAMS variables. If it is not one of these, the LOCAL name space and then the USER name space are examined. If the name is not found here, a TASK name space is examined; and if not found here, the SYSTEM name space is examined. All <*actual_name*> resolution follows this kind of bottom-up search through the hierarchical name space.

When names are created as the result of either a **isa** or **instantiates_a** statement, the name may be entered into the name space at any scope by means of a <*scope_clause*>, as in

```
<<    mary  instantiates_a  STUDENT
            scope is TASK           >>
```

The appropriate portion of the name space is examined to verify that the name, in this case *mary*, does not already exist. If it does, at the time a preprocessing, a preprocessor ERROR is generated. If by chance, between the time of preprocessing and execution, the name has been entered into the name space by a different process, a run-time failure occurs, and in the current version execution aborted with an ERROR message.

Because the same symbolic name, for example *type*, may appear in two different TASK partitions, say in *matrix* and *adams*, with different denotational meanings (e.g. one may denote

an attribute, while the other denotes a map), there is a possibility of ambiguity when it is used in an ADAMS statement. To resolve this ambiguity, a program containing ADAMS statements can only refer to one TASK partition. This is established when the code is first interpreted by the preprocessor. If, for example, *jlp* is compiling an *matrix* program he would use the command

```
adams -t matrix  <source_file_name>
```

If the *school* name space is to be used, the command

```
adams -t school  <source_file_name>
```

would resolve ADAMS names with respect to that name space. The simple preprocessor command

```
adams <source_file_name>
```

that we saw in Section 2.6, makes use of a default *adams* TASK name space to which all users are attached. Thus, at the time of preprocessing ADAMS code, a unique path through the name space is established, and the interpretation of all symbolic ADAMS names is resolved along this path.

All names visible along any particular path can be displayed by the utility *viewpath* , which also displays their scope and general nature, e.g. CLA(ss), COD(omain), or INS(tance). The same symbolic name may appear in different portions of the name space with different meanings. For completeness, we must note that the same name may be used to denote a class, a codomain, and an element instance, all within the same portion of the name space. ADAMS will differentiate on the basis of syntactic usage — however, we heartily disapprove of this practice!

The ability to enter the same name, but with different denotational meanings, at different levels of the name space can be quite powerful. We say that the lower level name **masks** the one at higher scope. This is why we were able to redefine the meaning of the codomain name REAL in the declaration of MATRIX classes in Figure 5-4. This definition of the REAL codomain was installed in the USER name space, and so became the definition that was used when the name REAL was resolved in subsequent code. But, name masking can also cause problems. We may want to use the name at the higher scope. If it is known that the desired meaning of an ADAMS name is one at a higher level in the name space and we suspect possible masking, the default bottom-up method of name resolution may be circumvented by using a **scoped name**, in which the desired scope preceeds the name. For example, if in the matrix applications we wanted to declare some real attributes of class R_ATTR with respect to the system defined REAL codomain, we could declare

```
<<    R_ATTR  isa  ATTRIBUTE
            with image  SYSTEM  REAL      >>
```

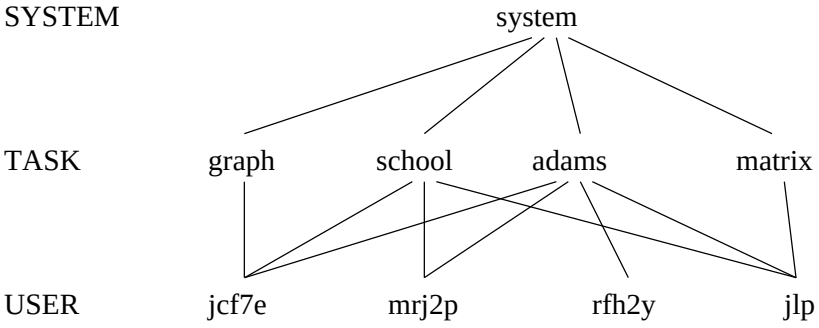
The programmer has considerable freedom to control the resolution of ADAMS names with their denotational meanings.

Because the ADAMS name space is dynamic, name masking can lead to other problems. Entry of a persistent name at one scope, may mask another at higher scope that had hitherto been used in run-time lookup. A working program may begin to fail. Similarly, erasure of a name may **unmask** a name at higher scope, resulting in erratic behavior. Rescoping a name from one scope to another can also cause both masking and unmasking problems. ADAMS can determine when entry, erasure, or rescoping of a name will change the name masking along a visibility path, and it can broadcast a message to all users who might be affected.†[22] But, it cannot prevent compiled code from behaving anomalously. This characteristic of dynamic, persistent name spaces remains an open research issue.

---

†[22] This is not currently implemented.

The structure of the ADAMS name space is also dynamic. Any ADAMS user can create a new subspace at the shared TASK scope with the command *newtask*. For example, *jcf7e* might create the task *graph* to store the class declarations of various graph procedures, so that the persistent name space now would look like




Persistent ADAMS name space
after *jcf7e* creates a new *graph* task
Figure 6-2

No user "owns" a task, nor any of the data denoted by names in a task or *system* subspace. These names, and this data, are by definition "shared". In its current incarnation, any ADAMS user may invoke *addtask* to make the data names, and data elements, of any task at available to his procedures. In a more widely distributed, commercial version some control, based on permissions, must be incorporated in the *addtask* facility. Currently, the only implemented permission regards name erasure. Shared names may only be erased by their creators.

## 6.2. Name Space Utilities

There are a number of command line utilities that can be used to view and manipulate the name space. As ADAMS evolves, we expect there will be more. The existing utilities are:

| | |
|---|---|
| newuser | - Create a new USER name space |
| newtask | - Create a new TASK name space partition |
| addtask | - Add a user to a task, i.e. make it visible |
| | |
| namespace | - Display all portions of the name space visible to the user |
| viewpath | - Display the name space visible along a single path |
| | |
| warmstart | - Erase selected entries in the name space, made by the user |

The *warmstart* utility is of considerable value when developing new applications. It provides for the selective erasure of names, especially class declarations, that have been made by the user that are incorrect for some reason. However, like all erasure, it does not delete any associated structures and so may leave unaccessible detritus in the ADAMS data space.

## 7. Syntax of ADAMS

This section presents the formal syntax of ADAMS as a phrase structure grammar, such as might be used by a parser like *yacc* [Joh75], to accept ADAMS statements. To assure correctness, it is, in fact, the *yacc* grammar that our preprocessor currently uses. Those constructs which will be parsed correctly, but for which no code generation is as yet provided, are again marked with †.

### 7.1. Lexical Tokens

The following is a table of the tokens of the language, together with the source code strings that generate them. All reserved ADAMS words, such as *intersect*, *for_each*, *isa*, are included here, together with certain kinds of delimited strings.

We would note that the current version of ADAMS is case insensitive; the source code strings *for_each*, *FOR_EACH*, and *For_Each* will all be interpreted as the FOR_EACH token. In some cases, other alternative strings, such as *for_all*, may also be recognized as a token. These alternative strings have been indicated.

| lex TOKEN | Corresponding String (all are case insensitive!) |
|---|---|
| ABORT | abort |
| A_UID_OF | A_uid_of |
| ADAMS_PROC | ADAMS_procedure |
| ADAMS_VAR | ADAMS_var |
| †ALL | all |
| AND | and |
| ASSIGN_OP | <- |
| ATTRIBUTE | ATTRIBUTE |
| ATTRIBUTES_OF | attributes_of |
| BAR | \| |
| CARDINALITY | cardinality |
| <string>CHARS | [a-zA-Z0-9]+ |
| CLASS_OF | class_of |
| CLASS_TOKEN | CLASS |
| CLOSE | close_ADAMS |
| CODOMAIN | CODOMAIN |
| COLON | : |
| COMPLEMENT | complement |
| CONSISTING | consisting (or consisting of) |
| DELETE | delete |
| <string>DIGITS | [0-9]+ |
| DO | do |
| ELEMENTS | elements |
| ELEMENT_OF | element_of |
| END_ADAMS_PROC | end_ADAMS_procedure |
| EQUAL | = |
| ERASE | erase |
| EXIT_LOOP | exit_loop |
| †EXISTS | exists |
| FOR_EACH | for_each (or for_all) |
| FORWARD | forward |
| †<string>FREE_VAR | @[a-zA-Z] |
| FROM | from |
| GREATER_EQ | >= |
| GREATER_THAN | > |
| HAVING | having |
| <string>HOST_EXPR_STR | @[a-zA-Z0-9 _+*/-%]*@ |
| <string>HOST_PARAM_STR | $[a-zA-Z0-9 {}()[]&,_:.\|+*/-"0]*$ |
| IMAGE | image |
| INSERT | insert |

| | |
|---|---|
| INSTANTIATES_A | instantiates_a |
| IN | in |
| INTERSECT | intersect |
| INTO | into |
| INVOKE | invoke |
| ISA | isa |
| L_CURL | { |
| LESS_EQ | <= |
| LESS_THAN | < |
| <string>LITERAL_VALUE | ’[#"a-zA-Z0-9_,;:@!~ {}()[]<>?.|$+*/-%’ |
| *or* | "[#’a-zA-Z0-9_,;:@!~ {}()[]<>?.|$+*/-%" |
| †LOCK | lock |
| L_PAREN | ( |
| L_SQUARE | [ |
| LOCAL_TOKEN | LOCAL |
| MAP_TOKEN | MAP |
| MAPS_OF | maps_of |
| NAME_OF | name_of |
| NOT_EQ | != |
| NULLSET | nullset |
| OR | or |
| OPEN | open_ADAMS |
| †<string>PARAM | $[1-99] |
| PERIOD | . |
| PROTOTYPE | ADAMS_prototype |
| PROVIDED | provided |
| R_CURL | } |
| R_PAREN | ) |
| R_SQUARE | ] |
| †RANDOM | random |
| RANGE | .. |
| REMOVE | remove |
| †RESCOPE | rescope |
| <string>REG_EXPR | #["’a-zA-Z0-9_,;:@!~ {}()[]<>?.|$+*/-%# |
| SEMI | ; *NOTE, ; is illegal in any ADAMS statement (except string)* |
| SCOPE | scope (or scope is) |
| SET_TOKEN | SET |
| SET_ASSOC_OP | -> |
| STAR | * |
| STMT_BEGIN | << |
| STMT_END | >> |
| SYSTEM_TOKEN | SYSTEM |
| TASK_TOKEN | TASK |
| TREAT_AS | treat_as |
| †TR_START | tr_start |
| †TR_END | tr_end |
| <string>TYPE | int, float, double, char*, or UID |
| UDF | udf |
| UKN | ukn |
| UNDER | _ |
| UNION | union |
| †UNLOCK | unlock |
| USER_TOKEN | USER |
| VAR | var |

## 7.2. Grammar Productions

The following productions have been stripped from the *yacc* parser, and slightly modified for readability, and grouped according to their general purpose. All presume the tokens of the preceding section.

```
adams_body:        ∈                              empty body
                 | adams_body adams_stmt

adams_stmt:        STMT_BEGIN  attr_decl_stmt STMT_END
                 | STMT_BEGIN  a_proc_decl_stmt STMT_END
                 | STMT_BEGIN  a_proc_end_stmt  STMT_END
                 | STMT_BEGIN  assign_stmt STMT_END
                 | STMT_BEGIN  class_decl_stmt STMT_END
                 | STMT_BEGIN  close_stmt STMT_END
                 | STMT_BEGIN  codomain_decl_stmt STMT_END
                 | STMT_BEGIN  element_inst_stmt STMT_END
                 | STMT_BEGIN  exit_loop_stmt STMT_END
                 | STMT_BEGIN  insert_stmt STMT_END
                 | STMT_BEGIN  a_proc_invoke_stmt STMT_END
                 | STMT_BEGIN  looping_stmt STMT_END
                 | STMT_BEGIN  map_decl_stmt STMT_END
                 | STMT_BEGIN  open_stmt STMT_END
                 | STMT_BEGIN  a_prototype_stmt STMT_END
                 | STMT_BEGIN  remove_stmt STMT_END
                 | STMT_BEGIN  set_decl_stmt STMT_END
                 | STMT_BEGIN  ADAMS_var_stmt STMT_END
```

## Open, Close Statements:

```
open_stmt:         OPEN   actual_name

close_stmt:        CLOSE   actual_name
```

## Codomains:

```
codomain_decl_stmt:   dict_class_entry ISA CODOMAIN  membership_clause
                          cod_decl_options

membership_clause:    CONSISTING  REG_EXPR

cod_decl_options:     ∈                    empty options
                    | cod_decl_options  cod_decl_opt

cod_decl_opt:         undefined_clause
                    | unknown_clause
                    | treat_as_clause
                    | scope_clause

treat_as_clause:      TREAT_AS  actual_name

undefined_clause:     UDF  EQUAL  literal

unknown_clause:       UKN  EQUAL  literal

codomain_name:        actual_name
                    | param_seg
                    | scope codomain_name
```

Class and Element Declarations:

```
class_decl_stmt:        dict_class_entry ISA super_class_list
                            class_decl_body

class_decl_body:        ∈                           empty class declaration
                      | class_decl_options

class_decl_options:     class_decl_opt
                      | class_decl_options class_decl_opt

class_decl_opt:         scope_clause
                      | association_clause
                      | restriction_clause
                      | FORWARD

association_clause:     HAVING set_desig
                      | HAVING actual_name EQUAL set_desig

†restriction_clause:    PROVIDED L_SQUARE predicate R_SQUARE

super_class_list:       CLASS_TOKEN
                      | class_name
                      | super_class_list AND class_name

element_inst_stmt:      dict_inst_entry INSTANTIATES_A class_name
                            opt_scope_clause

opt_scope_clause:       ∈                           empty clause
                      | scope_clause
```

Attribute and Map Declarations:

```
attr_decl_stmt:         dict_class_entry ISA ATTRIBUTE
                            IMAGE codomain_name
                            class_decl_options
                      | dict_class_entry ISA ATTRIBUTE
                            IMAGE codomain_name

map_decl_stmt:          dict_class_entry ISA MAP_TOKEN
                            IMAGE class_name
                            class_decl_options
                      | dict_class_entry ISA MAP_TOKEN
                            IMAGE class_name
```

Set Declarations, Operators, and Loops:

```
set_decl_stmt:          dict_class_entry ISA SET_TOKEN set_type ELEMENTS
                            class_decl_options
                      | dict_class_entry ISA SET_TOKEN set_type ELEMENTS

insert_stmt:            INSERT element_desig INTO set_desig

remove_stmt:            REMOVE element_desig FROM set_desig

looping_stmt:           loop_prefix loop_body

loop_prefix:            FOR_EACH ADAMS_var IN set_expr DO

loop_body:                  %prec error             empty body is error
                      | adams_stmt
```

```
                    | loop_body  adams_stmt

exit_loop_stmt:       EXIT_LOOP

set_type:             class_name
                    | ATTRIBUTE
                    | MAP_TOKEN
                    | SET_TOKEN
```

## Assignment Operators and Expressions:

```
assign_stmt:          host_var   ASSIGN_OP  value_desig
                    | left_side  ASSIGN_OP  host_var
                    | left_side  ASSIGN_OP  literal
                    | left_side  ASSIGN_OP  element_desig
                    | left_side  ASSIGN_OP  set_expr

left_side:            element_name
                    | dot_expr

dot_expr:             element_desig  PERIOD  element_name
                    | †free_var PERIOD element_name

set_expr:             set_desig
                    | L_PAREN  set_expr  R_PAREN
                    | set_expr  COMPLEMENT  set_expr
                    | set_expr  INTERSECT  set_expr
                    | set_expr  UNION  set_expr
```

## Predicates:

```
predicate:            disjunct
                    | predicate OR disjunct

disjunct:             conjunct
                    | disjunct  AND  conjunct

conjunct:             term
                    | L_PAREN predicate R_PAREN
                    | †quantifier L_SQUARE predicate R_SQUARE

term:                 domain_comparison
                    | element_desig  EQUAL   element_desig
                    | element_desig  NOT_EQ  element_desig
                    | †free_var  EQUAL   element_desig
                    | †free_var  NOT_EQ  element_desig

domain_comparison:    value_desig  comparator  value_desig  comparator value_desig
                    | host_var     comparator  value_desig  comparator host_var
                    | value_desig  comparator  value_desig  comparator host_var
                    | host_var  comparator  value_desig  comparator value_desig
                    | value_desig  comparator  value_desig
                    | host_var     comparator  value_desig
                    | value_desig  comparator  host_var

comparator:           EQUAL
                    | NOT_EQ
                    | LESS_THAN
                    | LESS_EQ
                    | GREATER_THAN
                    | GREATER_EQ
```

†quantifier: L_PAREN ALL bound_var IN set_desig R_PAREN
| L_PAREN EXISTS bound_var IN set_desig R_PAREN

†free_var: FREE_VAR

bound_var: ADAMS_var

## Element, Set, and Value Designators:

element_desig: element_name
| dot_expr

set_desig: element_desig
| enumerated_set
| retrieval_set
| class_name SET_ASSOC_OP actual_name
| NULLSET

enumerated_set: L_CURL enumeration_list R_CURL
| L_CURL R_CURL *empty enumerated set*

enumeration_list: enumeration_element
| enumeration_list enumeration_element

enumeration_element: actual_name *may also be ADAMS_var*
| scope actual_name
| subscripted_name_range
| enumeration_element PERIOD enumeration_element

retrieval_set: L_CURL bound_var IN set_expr BAR predicate R_CURL

value_desig: dot_expr
| literal

literal: LITERAL_VALUE

## ADAMS Names:

char_seg: CHARS

param_seg: PARAM

actual_name: actual_name UNDER char_seg
| actual_name UNDER DIGITS
| char_seg

†param_name: param_seg
| actual_name UNDER param_seg
| param_name UNDER CHARS
| param_name UNDER DIGITS
| param_name UNDER param_seg

class_name: actual_name
| scope actual_name
| VAR actual_name

host_var: BAR host_var_name specifier BAR

host_var_name: actual_name
| actual_name PERIOD actual_name
| STAR actual_name PERIOD actual_name

```
specifier:              var_type
                      | var_type  DIGITS

var_type:               TYPE

subscripted_name:       actual_name  L_SQUARE  subscript_list  R_SQUARE
                      | scope  actual_name  L_SQUARE  subscript_list  R_SQUARE

subscripted_name_range: actual_name  L_SQUARE  range_subscript_list  R_SQUARE

range_subscript_list:   range_subscript_list  range_subscript
                      | range_subscript

range_subscript:        subscript_value  RANGE  subscript_value
                      | subscript_value

subscript_list:         subscript_value
                      | subscript_value  subscript_list

subscript_value:        integer
                      | host_var_name
                      | HOST_EXPR_STR
                      | †param_seg

subscript_pattern:      actual_name L_SQUARE subscript_count R_SQUARE

subscript_count:        STAR
                      | subscript_count STAR

integer:                DIGITS

ADAMS_var:              actual_name

ADAMS_var_list:         ADAMS_var
                      | ADAMS_var_list ADAMS_var

ADAMS_var_stmt:         ADAMS_VAR  ADAMS_var_list

element_name:           actual_name                    may also be ADAMS_var
                      | scope  actual_name
                      | VAR  actual_name
                      | subscripted_name
                      | reserved_attr_name

reserved_attr_name:     NAME_OF
                      | CLASS_OF
                      | CARDINALITY
                      | ATTRIBUTES_OF
                      | MAPS_OF
                      | ELEMENT_OF
                      | †RANDOM
                      | A_UID_OF
```

## ADAMS Name Space:

```
dict_class_entry:     actual_name
                    | †param_name
                    | VAR actual_name
                    | †VAR param_name

dict_inst_entry:      actual_name
                    | VAR actual_name
                    | subscript_pattern

scope_clause:         SCOPE scope

scope:                SYSTEM_TOKEN
                    | TASK_TOKEN
                    | USER_TOKEN
                    | LOCAL_TOKEN
```

## ADAMS Procedures:

```
a_prototype_stmt:     PROTOTYPE proc_name L_PAREN class_prototypes R_PAREN

class_prototypes:     class_list  host_param_list
                    | class_list
                    | host_param_list

class_list:           class_name
                    | class_name  class_list

a_proc_decl_stmt:     ADAMS_PROC proc_name L_PAREN formal_proc_params R_PAREN

formal_proc_params:   a_formal_param_list  host_param_list
                    | a_formal_param_list
                    | host_param_list

a_formal_param_list:  a_formal_param
                    | a_formal_param  a_formal_param_list

a_formal_param:       class_name  actual_name

host_param_list:      HOST_PARAM_STR

a_proc_end_stmt:      END_ADAMS_PROC

a_proc_invoke_stmt:   INVOKE proc_name L_PAREN actual_proc_params R_PAREN

actual_proc_params:   a_actual_param_list  host_param_list
                    | a_actual_param_list
                    | host_param_list

a_actual_param_list:  a_actual_param
                    | a_actual_param  a_actual_param_list

a_actual_param:       element_desig

proc_name:            actual_name
```

## 8. References


[Bar89] P. Baron, The ADAMS Preprocessor, IPC TR-89-009, Institute for Parallel Computation, Univ. of Virginia, Dec. 1989.

[Che76] P. P. Chen, The Entity-Relationship Model---Toward a Unified View of Data, *Trans. Database Systems 1*,1 (Mar. 1976), 9-36.

[Cle91] T. P. Cleary, A Relational Interface to an Object Based System, or Translating SQL to ADAMS, IPC TR-91-009, Institute for Parallel Computation, Univ. of Virginia, Aug. 1991.

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

[Jan89] S. A. Janet Jr., The ADAMS Storage Management System, IPC TR-89-008, Institute for Parallel Computation, Univ. of Virginia, Aug. 1989.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler Compiler, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, NJ, Sep. 1975.

[KeR88] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, second edition 1988.

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

[Klu88] C. Klumpp, Implementation of an ADAMS Prototype: the ADAMS Preprocessor, IPC TR-88-005, Institute for Parallel Computation, Univ. of Virginia, Aug. 1988.

[LeS75] M. E. Lesk and E. Schmidt, Lex — A Lexical Analyzer Generator, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, NJ, Oct. 1975.

[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.

[Pfa77] J. L. Pfaltz, *Computer Data Structures*, McGraw-Hill, Feb. 1977.

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

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

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

[PFG89a] J. L. Pfaltz, J. C. French, A. Grimshaw, S. H. Son, P. Baron, S. Janet, Y. Lin, L. Loyd and R. McElrath, Implementation of the ADAMS Database System, IPC TR-89-010, Institute for Parallel Computation, Univ. of Virginia, Dec. 1989.

[PFG89b] J. L. Pfaltz, J. C. French, A. Grimshaw, S. H. Son, P. Baron, S. Janet, A. Kim, C. Klumpp, Y. Lin and L. Loyd, The ADAMS Database Language, IPC TR-89-002, Institute for Parallel Computation, Univ. of Virginia, Feb. 1989.

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

[PFG91] J. L. Pfaltz, J. C. French and A. Grimshaw, An Introduction to the ADAMS Interface Language: Part I, IPC TR-91-06, Institute for Parallel Computation, Univ. of Virginia, Apr. 1991.

[Pfa92] J. L. Pfaltz, Programming over a Persistent Data Space, IPC TR-92-008, Institute for Parallel Computation, Univ. of Virginia, Sep. 1992.

[PfF93] J. L. Pfaltz and J. C. French, Scientific Database Management with ADAMS, *Data Engineering 14*,1 (Mar. 1993).

[Rie90] E. Rietscha, *The ADAMS Interactive Interpreter*, MCS Project, Univ. of Virginia, Dec. 1990.

[Wat90] G. Watson, *Embedding ADAMS in Fortran*, MCS Project, Univ. of Virginia, Dec. 1990.

## 9. Index of Terms

**Table of Contents**