DOE/ER/25063--T7

DE89 010926

Implementation of an ADAMS Prototype: the ADAMS Preprocessor (AP)

Cathleen L. Klumpp
John L. Pfaltz

IPC-TR-88-005
August 9, 1988

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22901

This research was supported in part by JPL Contract #957721.

DISCLAIMER

Abstract

This report describes the implementation of an ADAMS prototype called the ADAMS Preprocessor (AP). The goals of the project are discussed and then the specific subset of the ADAMS interface language which is supported by the AP is defined. After presenting the system's user interface and an overview of the language's basic concepts, a BNF notation is used to precisely define its syntax and the exact semantics of each statement are specified. In addition, complete sample ADAMS programs are supplied. Finally, the results of the prototype effort are summarized. This document is intended to serve as a user's guide for those wishing to use the AP to implement ADAMS programs. However, readers who are interested only in a high level overview of the prototype and its capabilities may skip the more detailed sections.

TABLE OF CONTENTS

1. Introduction

The Advanced DAta Manipulation System (ADAMS) is being developed by the Institute for Parallel Computation at the University of Virginia. This paper describes the implementation of a prototype for this system called the ADAMS Preprocessor (AP). Before defining the objectives of the AP, a brief overview of the ADAMS project is given.

The ADAMS goal is to provide a *generic* facility which supports the creation of databases, the persistent storage of data in those databases, and the retrieval of stored data. Unlike a traditional database management system (DBMS), the ADAMS facility is generic because the underlying representation of an ADAMS database is independent of any particular database model (e.g., relational, network). Also unlike a typical DBMS, no interactive user support capabilities are provided by ADAMS. A major objective in developing ADAMS is to separate the model viewed by the end user from the actual database implementation scheme. Any traditional DBMS, regardless of its user interface model, can then be implemented using ADAMS as a fundamental data storage and retrieval utility. The many advantages of such an approach to database implementation are discussed in [Pf87] and [PfF]. One significant advantage involves the goal of effectively implementing ADAMS on a parallel, hypercube computer. The ADAMS approach allows for more efficient implementation on such an architecture. To support its generic facilities, ADAMS must provide a mechanism by which an application (presumably a DBMS, but possibly any program) can interface with an ADAMS database. The ADAMS embedded language supplies such an interface by allowing an application program written in a high level programming language, known as the *host language*, to create and access persistent ADAMS databases.

The purpose of developing the AP was to implement the ADAMS interface language, as presented in the preliminary design document [Pf87]. The major goal of the project was to investigate the feasibility of proposed ADAMS concepts. The AP allows ADAMS programs to be written and tested, thus supporting continued exploration of the specific language features and of the overall approach. As is typical of prototypes, the system's functionality was of more importance than the efficiency of its implementation. Because the methods used to implement the interface language were not relevant, it was not developed on a parallel machine. However, the ADAMS design is independent of the machine architecture on which it is implemented, so future versions of ADAMS can easily be developed to take advantage of a parallel environment. Because the objective was to build the prototype as rapidly as possible, the AP supports only a subset of the ADAMS language. However, this is not a serious limitation because the subset includes the significant features necessary to accomplish the goal of exploring fundamental ADAMS capabilities.

The remainder of the paper describes the AP implementation in detail, including its user interface and the subset of the ADAMS interface language it supports. This document is intended to serve as a user's guide for those wishing to use the AP to implement ADAMS programs. Readers who are interested only in a high level overview of the prototype and its capabilities may skip the following: sections 2.1 and 2.2, all of section 4, and all of section 5.

2. User Interface

In designing the user interface of the ADAMS prototype, two alternatives were available. The first possibility was to build an *interpreter,* an interactive tool which executes inputed ADAMS source statements immediately without compilation. The second alternative was a non-interactive *preprocessor* which translates *embedded* ADAMS source statements into equivalent statements in the host language. These statements are then compiled and linked with the necessary ADAMS utilities and user supplied host code to produce an executable program. Since ADAMS is designed to function as an embedded database "back end" in conjunction with programs written in a host computational language, the preprocessor approach seemed most appropriate. In addition, it better fulfilled the established requirements for rapid prototyping because it avoids the development of an independent, interactive front end.

To use the preprocessor, a user must first create an ADAMS *source file.* This file contains a combination of valid ADAMS and host language statements. The current implementation of the AP uses C as the host language. However, other host language (e.g., FORTRAN, Pascal) preprocessors will eventually be available as well. The AP is used to generate an equivalent host language file by translating all ADAMS source statements into one or more calls to AP utilities. Statements already in the host language are simply copied without any translation. The *generated file* is then compiled and linked, resulting in an *executable program.* Finally, this program is executed.

2.1. Using the AP

This section is intended for those who actually plan to use the AP software. If the reader does not belong in this category, it may be skipped. The AP user may use the system to perform the following actions.

Access AP Utilities

The AP code is currently located on the following UVA CS Department computer which runs under the UNIX 5.0 operating system: "babbage". In order to access any of the AP utilities, the user must add the following path name to his default UNIX search paths: "/va1/clk3h/gra/new_code".

Initialize AP Environment

The environment necessary to run the AP is created using the "adams_init" utility. It creates the directories and files needed to support persistent ADAMS databases. Note that initialization only needs to be done *once* for each directory in which the user wishes to run the AP.

invocation: adams_init [CR]

Re-initialize AP Environment

While testing an ADAMS program, the user will often wish to completely re-initialize the AP environment (i.e., delete all existing persistent data). To do so, use the "adams_rinit" utility, which first removes all existing database files and then runs the "adams_init" utility.

invocation: adams_rinit [CR]

Create Generated File

After proper initialization, the "adams_gen" utility is used to translate an ADAMS source file (containing both ADAMS and C statements) into an equivalent generated C file. This utility

has one argument <fn>, the name of the source file. Note that this file must be named with a ".src" extension and that the extension is not included in the <fn> argument. Also note that this source file must be located in the directory from which the "adams_gen" utility will be invoked. There are a few requirements regarding the contents of the source file. They are explained in the "template.src" file located in the "/va1/clk3h/gra/new_code" directory. This file can be used as a template when creating a new ADAMS source file. If the generation is successful, a file with the name <fn> and the extension ".c" is created and placed in the current directory.

invocation: adams_gen <fn> [CR]

Create Executable Program

After successful generation, the "adams_exec" utility is used to compile the newly created C file and link it with the necessary AP utilities, creating an executable program. The <fn> argument must be the same name given in the previous generation phase. Note that it is assumed that the generated file <fn>.c exists in the current working directory. If both the compilation and linking phases are successful, a file with the name <fn> and the extension ".x" is created and placed in the current directory.

invocation: adams_exec <fn> [CR]

Execute Program

After successful creation of the executable program, the program may be run. Any modifications to the ADAMS permanent databases are not made until the entire program has executed successfully. Thus, if a program terminates as the result of an error, the session is aborted.

invocation: <fn>.x [CR]

2.2. Sample AP Session

The scenario in Figure 1 is intended to help an actual AP user who is just getting started with the system. Assume that the file "test.src" is a source file containing ADAMS statements embedded in a C program. Also, assume that this file is the only one existing in the current directory. Commands issued by the user are shown in bold type.

The directory is first initialized. Then the AP is used to generate the equivalent host language file from the ADAMS source file and to create the executable program. Finally, the program is executed and it prints a message indicating a successful run. In the example, the UNIX "ls" command is used before and after each AP command to show what files and/or directories are created during its execution.

```
babbage $ ls
test.src
babbage $ adams_init
babbage $ ls
conv_func.dict
dict
test.src
babbage $ adams_gen test

Parsing the source file...

Finished parsing.
babbage $ ls
conv_func.dict
dict
test.c
test.src
babbage $ adams_exec test
   cc -c test.c
   cc /val/clk3h/gra/new_code/code/ap_cplus.o ...
babbage $ ls
conv_func.dict
dict
test.c
test.o
test.src
test.x
babbage $ test.x

The ADAMS test program ran correctly!

babbage $
```

Figure 1. Sample AP Session

3. Language Overview

Before presenting the grammar for the ADAMS subset supported by the AP, it is useful to give a brief overview of its major language features. Note that this is a description of the language as it is implemented by the AP, and thus does not include some features present in the complete ADAMS system. For a more complete definition of the actual ADAMS interface language refer to [Pf87].

3.1. Primitive Types: The Co-domain

The database-specific nature of ADAMS has a great impact on the primitive types supported. Specifically, because ADAMS is an embedded interface language supporting only data storage and retrieval, there is no need to differentiate between the various standard primitive types (e.g., integer, real, boolean). Only the host computational language must support such types. Therefore, all *data values* are treated uniformly as ASCII strings. The ADAMS term *co-domain* is used to denote this single primitive type.

Although all ADAMS data values are represented as strings, user defined co-domains differentiate between various semantic groupings of strings. For example, a co-domain may identify all strings which represent a particular data type, such as all integers or a specific subrange of integers. A co-domain definition includes a *regular expression* that specifies all strings (data values) belonging to the type it represents. Functions which convert a co-domain value from the ADAMS representation into the equivalent host language representation *(fetch)* or vice versa *(store)* may be declared when defining a co-domain. These functions are written by the user in the host language and are automatically invoked by the system whenever data is transferred between ADAMS and the host language (see section 3.4).

3.2. Type Constructors: The Class

The fundamental ADAMS type constructor is the *class*. A class represents a group of objects which all share common characteristics (i.e., have the same type). Each member of a class is known as an *instance* of the class. To support the modeling of complex relationships among data objects, the ADAMS language supports *inheritance*. A superclass inherits the properties of an existing class known as its *parent*. In addition, it may have some of its own unique characteristics. The current version of the AP supports *single* (as opposed to multiple) inheritance. That is, each class may have exactly one immediate parent. Thus, class definitions in ADAMS form a tree structure, or *class hierarchy*, rather than a network.

There are three predefined *system classes* in ADAMS: *attributes*, *maps*, and *sets*. These definitions include operations and properties associated with the instances of each class. In addition to these system classes, *user defined classes* are allowed. All ADAMS types are constructed from some combination of predefined and user defined classes.

Attributes

Members of this class represent *functions* mapping a given class instance into a given co-domain. For an attribute to be defined on a specific instance, it must be *associated with* the instance's class. This practice of associating particular functions with a class is explained later in this section.

Attribute instances are created by declaring the co-domain which is to serve as the attribute's *image* (i.e., the type of the data value returned by the application of the attribute function). The method by which an attribute's value is obtained is also specified. This version of the AP supports only explicit assignment of attribute values (although ADAMS itself postulates other

methods). A user selected string signifying an undefined value for the attribute may also be supplied. Otherwise, the literal string value "undefined" is used by default.

An *application operation* is defined for attributes. When an attribute is applied to an instance whose class is associated with that attribute, the correct value from the attribute's image is returned.

Maps

The map class is similar to the attribute class because map instances also represent functions. However, maps link an instance of one class with an instance of another. For a map to be defined on a given instance, an association between the map and the instance's class must exist.

Map instances are created in the same manner as attributes, except that a map's image is another class rather than a co-domain, and no undefined value may be specified. A map is applied to an instance with the same application operator.

Sets

The set class is one of the most important concepts in ADAMS because sets provide the basic type constructor used to model various database structures. Instances of this class represent sets of instances belonging to some other common class. A set class is characterized by the class of the instances its members may contain. Note that the selection of such a class is virtually unrestricted, so sets may themselves contain sets, which can in turn contain attributes, maps, or instances of some other class. Thus, the ability to build complex and powerful types in ADAMS is supported largely by the set concept. Set instances are created by specifying the desired set class. An optional list of initial set members may be given.

User Defined Classes

A user defined class may be specified by declaring its parent class. There are no predefined operations associated with such a class. An instance of a user defined class is created simply by declaring the desired class name.

Association and Restriction

Both sets and user defined class definitions can include *association* clauses. These clauses associate particular attributes and/or maps with the class being defined. The associated functions will then be defined on all instances of the class. In a similar manner, both types of class definitions can include *predicate restrictions* which enforce desired integrity constraints on the members of the class. Although an abbreviated form of such restrictions may be declared using the current version of the AP, they are not actually used in validation.

3.3. Data Persistence: The Dictionary

The ADAMS concept of *scope* characterizes the persistence of ADAMS database *elements*. An element is one of the following: a co-domain definition, a class definition, or a class instance. Access to all ADAMS database elements is by means of a *dictionary*, with a separate dictionary existing for each possible scope. When a named element is created, the scope specified by the user determines the dictionary into which its name is entered. Elements which have a *local scope* are not persistent and, therefore, disappear when the program that created them terminates. Elements which have a *user scope* are visible only to the user who created them and are recorded permanently in a local user dictionary. The *system scope* provides a dictionary of permanent objects common to all ADAMS users. Users may access elements stored in this dictionary, but

they cannot be modified. The predefined system classes (e.g., attribute, map, set) discussed above have this scope. The user may specify the desired scope whenever a co-domain or class is defined, or when a class instance is created. If a particular scope is not given when such a dictionary entry is made, the scope is local by default.

3.4. Data Manipulation

The ADAMS language supports the following facilities for accessing and manipulating class instances stored in an ADAMS database. Note that uniform access is provided for both persistent (user or system scope) and non-persistent (local) instances.

Assigning and Retrieving Attribute Values

Attribute values of an existing instance may be directly assigned (from other attribute values) using the *assign statement*. In addition, they may be assigned from host language variables and constants using the *store statement*. If a store conversion function was declared in the attribute's co-domain definition (see section 3.1), then it is automatically invoked before the value is stored to convert between the internal representations of the host and ADAMS languages. Attribute values are returned to host variables using the *fetch statement*. If a fetch conversion function exists (see section 3.1), it is called before returning the value. If no store (or fetch) conversion function was declared, then the type of the host variable being stored from (or fetched into) must be a character string.

Assigning Maps

The *assign statement* (when applied to maps) assigns a value to a map of an existing instance. The new value must be another existing instance (which is a member of the map's image class). Note that the instance being assigned as the new map value is not copied (i.e., the map acts as a pointer to instance).

Assigning Instances

The *assign statement* (when applied to instances) assigns all the attribute and map values of an existing instance by copying the corresponding values from another instance. All inherited function values are included in the copy. If the instance being assigned is a set, then the current members of the set being copied are inserted into it. Note that copies are not actually made of each member instance. Only the structure which keeps track of the set's members is copied. Thus, inserting or deleting from the original will not affect the membership of the copy (and vice versa).

Modifying and Accessing Sets

The *insert statement* adds a new member to an existing set instance. Note that no copy of the instance being inserted is made (i.e., the insertion is made "by reference"). The *delete statement* removes a member from an existing set instance. Note that the instance being deleted still exists in the database, it is simply no longer a member of the set. The *for loop* allows sequential access to each member of an existing set instance. The body of the loop may contain both ADAMS and host statements, either of which may reference the current set member.

3.5. Dictionary Interrogation

The interface language, as implemented by the AP, allows the user to interrogate ADAMS dictionaries to obtain useful information about the elements stored in them. Two methods of interrogation are possible. Unlike the typical ADAMS statement (which is a separate statement

delimited by special begin and end markers), the first dictionary interrogation method allows a limited number of facilities to be called in the same manner as standard host language functions. Thus, they may appear in host language expressions (e.g., in the condition of an "if/then" statement or as an argument in a "printf" statement). The AP user may obtain the class of a given instance, the class of the members of a given set instance, the dictionary name of the set member currently being iterated on in a for all loop, the image class of a given map instance, and the internal id of a given instance. Also, a user may determine if a given name is currently entered in an ADAMS dictionary, if a given name represents an ADAMS class, and if a given instance is currently a member of a given set.

The second interrogation approach involves regular ADAMS statements. The *"view of " statement* searches the dictionary for all the attributes associated with a given class and assigns them to a given set. The *"map of " statement* searches the dictionary for all the maps associated with a given class and assigns them to a given set. These view sets may then be manipulated by the user in the same manner as any other set instance. Note that the views include all the associated attributes (or maps) because the search goes up the class hierarchy. The reader should see [PFW88] for a more in depth discussion of this topic.

4. Language Syntax

This section gives the grammar for the AP language subset. First the notation used to describe the syntax is given and then the syntax itself is defined. Finally, implementation notes relevant to the syntax are included.

4.1. Notation Conventions

An extended BNF notation, summarized in Figure 2, is used for the syntax definition. The symbol "xxx" represents any literal string of characters and an "e" denotes any legal BNF expression. All the items listed in the left column of Figure 2 are themselves legal expressions.. All of the grammar rules in the BNF are of the form "<x> ::= e". The ADAMS reserved words appear in the grammar rules as bold type.

Figure 3 identifies a special group of non-terminal symbols which all represent some literal string, and are thus all recognized by the same regular expression [a-zA-Z]*. Although they are the same syntactically, they differ in their associated semantics. Therefore, they are distinguished from each other in the BNF grammar rules to better express the appropriate meaning.

Expression	*Explanation*
'xxx'	terminal symbol
<xxx>	non-terminal symbol
[e]	optional occurrence of e
(e)	a grouping of syntactic elements
e+	one or more occurrences of e
e*	zero or more occurrences of e
e1 | e2	one occurrence of either e1 or e2

Figure 2. BNF Notation

<attr_name>	<loop_variable_name>
<co_domain_name>	<map_name>
<func_set_name>	<set_class_name>
<func_name>	<set_name>
<host_var>	<set_synonym>
<host_const>	<string>
<host_func>	<user_class_name>
<inst_name>	<variable>

Figure 3. Names Used in Syntax Definition

As mentioned in section 3.5, the ADAMS interface language does support some standard host language function calls which can be used to interrogate an ADAMS dictionary. In the syntax definition, the BNF format is not used to define these functions. Instead, a function declaration is given. First the type of the return value is given, then the name of the function and a list of the formal parameters (including their types). Since the AP host language is C, the "Boolean" type represents an "int" ("0" for false and "1" for true) and the "Charstr" type represents a "char*".

4.2. Syntax Definition

4.2.1. ADAMS Source File

All ADAMS source files consist of both ADAMS and host language statements. The <host_stmt> non-terminal symbol represents any valid host language statement (as determined by the language defintion), and thus is not defined further.

<adams_source> ::= ('<<' <adams_stmt> '>>' | <host_stmt>)+
<adams_stmt> ::= <co_domain_def> | <class_def> | <inst_creation> | <inst_manip>
<dict_interrog> | <name_var>

4.2.2. Co-domain Definition

<co_domain_def> ::= <co_domain_name> **is a CO_DOMAIN**
consisting of '#' <regular_expr> '#'
[<store_clause>]
[<fetch_clause>]
[<scope_clause>]
<store_clause> ::= **with store** <host_func>
<fetch_clause> ::= **with fetch** <host_func>

4.2.3. Class Definition

<class_def> ::= <user_class_def> | <set_class_def>

User Defined Class

<user_class_def> ::= <user_class_name> **is a** <user_class_desig>
[<class_def_clause>]
[<scope_clause>]
<user_class_desig> ::= <user_class_name> | <set_class_name> | **CLASS**

Set Class

<set_class_def> ::= <set_class_name> **is a SET**
of <class_desig> **elements**
[<class_def_clause>]
[<scope_clause>]

Association and Restriction

<class_def_clause> ::= (<assoc_clause> | <restrict_clause>)+
<assoc_clause> ::= **having** [<set_synonym> '='] <func_set>
<func_set> ::= <func_set_name> | <enum_func_set>
<enum_func_set> ::= '{' (<func_name>)+ '}'

<restrict_clause> ::= **in which** <quantifier> <predicate>
<quantifier> ::= '(' **for all** <variable_name> ')'
<predicate> ::= '[' <variable_name> '.' <attr_name> <rel_op> '"' <string> '"' ']'
<rel_op> ::= '==' | '!=' | '<' | '>' | '<=' | '>='

4.2.4. Instance Creation

<inst_creation> ::= <attr_creation> | <map_creation> | <user_creation> | <set_creation>

Attribute Class

<attr_creation> ::= <attr_name> **belongs to ATTRIBUTE**
with image <co_domain_name>
<value_assign_clause>
[<undef_clause>]
[<scope_clause>]
<undef_clause> ::= **with undefined** '"' <string> '"'
<value_assign_clause> ::= **value is assigned**

Map Class

<map_creation> ::= <map_name> **belongs to MAP**
with image <class_desig>
<value_assign_clause>
[<scope_clause>]

User Defined Class

<user_creation> ::= <inst_name> **belongs to** <user_class_name>
[<scope_clause>]

Set Class

<set_creation> ::= <set_name> **belongs to** <set_class_name>
[consisting of <set_value>]
[<scope_clause>]
<set_value> ::= <set_desig> | <enum_set>
<enum_set> ::= '{' (<inst_name>)+ '}'

4.2.5. Instance Manipulation

<inst_manip> ::= <attr_assign> | <store> | <fetch> | <map_assign> | <inst_assign>
| <set_access>

Attribute Assignment and Retrieval

<attr_assign> ::= **assign into** <attr_desig> **from** <attr_value>
<attr_value> ::= <attr_desig> | '"' <string> '"'

<store> ::= **store into** <attr_desig> **from** <host_value>
<host_value> ::= <host_var> | <host_const>

<fetch> ::= **fetch into** <host_var> **from** <attr_desig>

Map Assignment

<map_assign> ::= **assign into** <inst_desig> **from** <inst_desig>

Instance Assignment

<inst_assign> ::= **assign into** <inst_name> **from** <inst_desig>

Set Access

<set_access> ::= <insert> | <delete> | <for_all>
<insert> ::= **insert** <inst_desig> **into** <set_desig>
<delete> ::= **delete** <inst_desig> **from** <set_desig>
<for_all> ::= **for all** <loop_variable_name> **in** <set_desig> **do**
(<adams_stmt> | <host_stmt>)+

4.2.6. Dictionary Interrogation

<dict_interrog> ::= <attr_view> | <map_view>

<attr_view> ::= <set_desig> **view of** <class_name>
<map_view> ::= <set_desig> **map of** <class_name>

Host Language Functions

Charstr	class_of(inst : Charstr)
Charstr	class_of_element(set_inst : Charstr)
Charstr	dict_name_of(for_loop_var : Charstr)
Charstr	id_of(inst : Charstr)
Charstr	image_of(map_inst : Charstr)
Boolean	is_class(name : Charstr)
Boolean	is_dict_entry(name : Charstr)
Boolean	is_element_of(inst, set_inst : Charstr)

4.2.7. Miscellaneous

Designators

<attr_desig> ::= <inst_desig> '.' <attr_name>
<class_desig> ::= <user_class_name> | <set_class_name> | **ATTRIBUTE | MAP**
| SET | CLASS

<inst_desig> ::= <inst_name> [<map_desig>] [<map_desig>]
<map_desig> ::= '.' <map_name>
<set_desig> ::= <set_name> | <inst_desig>

Scope

<temp_scope> ::= **LOCAL**
<permanent_scope> ::= **USER**
<scope_clause> ::= **with scope** <permanent_scope>

Name Variables

<name_var> ::= **name_variable** (<inst_name> | <class_name>)+

4.3. Implementation Notes

regular expressions

When defining a co-domain, the user must surround the regular expression by "#" delimiters. The LEX definition of regular expressions is used by ADAMS to determine validity of such an expression. However, because the regular expression is not currently used by the AP implementation, a simple expression such as "#[a-z]#" may be used as a default. To satisfy the parser, some expression must be supplied by the user. Also, in the current implementation, blanks may not occur in the regular expression.

instance designators

In statements where a particular instance must be designated, nested maps may be used. However, this nesting is limited to only two levels. Thus, as noted in the grammar rule defining the non-terminal symbol <inst_desig>, an instance may be designated by any of the following forms:

(1) <inst_name>
(2) <inst_name>.<map_name>
(3) <inst_name>.<map_name>.<map_name>

In statements where a particular attribute is being designated, the attribute name may be added to the end of any of the above form of instance designator. Thus, the valid possibilities are:

(1) <inst_name>.<attr_name>
(2) <inst_name>.<map_name>.<attr_name>
(3) <inst_name>.<map_name>.<map_name>.<attr_name>

restriction clauses

Currently restriction clauses are not actually used to verify attribute values. However, they may be supplied in the class definition anyway, as eventually they will be used in validation.

specifying scopes

Currently the scope of a definition or instance creation may be specified by the user. The default scope is "LOCAL". If "USER" scope is indicated, an entry is made in the permanent user

dictionary. However, the AP does not allow the scope to be given within an ADAMS statement when referencing existing dictionary entries. Whenever a reference is made to an ADAMS name, the dictionaries are automatically searched in this order: LOCAL, USER, SYSTEM. The first entry found with the given name is used.

conversion functions

The current implementation does support the use of conversion functions in storing and fetching data. If such a function is supplied in the co-domain definition, it will be called every time a value in that co-domain is stored or fetched. These functions must satisfy the following constraints. There must be exactly one input parameter with type being character pointer. Also, the function must return a character pointer. For implementation reasons, the code for any conversion functions you use should be in a separate file and then included (using #include statement) with every ".src" file. This is required because the code must be compiled and linked every time, so that the functions are accessible whenever a store or fetch is carried out.

value assignment clause

Note that even though only one option is valid ("value is assigned"), this clause is currently required in the creation of attributes and maps. This requirement is somewhat inconvenient, but it must be followed anyway.

dictionary name variables

The name variable statement is a particulary useful feature supported by the AP. As defined in the grammar rule for the non-terminal symbol <name_var>, this statement is simply a list of names which will be used in the program as ADAMS variables. They are called "name" variables because they can be used in any ADAMS statement in place of a literal dictionary name. After name variables are declared, they may appear in an ADAMS statement wherever a dictionary name is expected. However, before being used in an ADAMS statement, the name variable must be declared as a host language character string and must be assigned some string value. In the BNF grammar rules, any non-terminal symbol of the form <xxx_name>, where the "xxx" represents any string, represents a dictionary name and may therefore be replaced by a name variable rather than a literal name.

The great advantage of these variables is that the same ADAMS statement may be executed over and over, but with different values assigned to the variables each time. Thus, ADAMS statements may more easily be placed in host language procedures. The name variables may be used as formal parameters for these procedures. An example of such a use is given in Figure 4. Whenever it is invoked, this C function creates a new instance of the "PERSON" class which is saved in the permanent database. The name of the new instance, which is used to make an entery in the ADAMS dictionary, is determined by the current value of the name variable "X". Thus, in this example, two different people (Cathy and Joe) are created.

Note that the current implementation requires that only one "name_variable" statement may appear during a single ADAMS execution. Thus, the user should collect all name variables and declare them in a single statement at the very beginning of the program.

```
<< name_variable X >>

main
{
    char X[30];

    DICT_init();

    create_person("Cathy McCabe");
    create_person("Joe Smith");

    DICT_close();
}

create_person(X)
  char* X;
{
    << X belongs to PERSON, with scope USER     >>
}
```

Figure 4. Using Name Variables

host language interrogation functions

When invoking one of the dictionary interrogation functions, a literal ADAMS dictionary name must be surrounded with double quotes when used as an argument. However, an ADAMS name variable (declared using the "name_variable" statement) does not need quotes. The "for all" loop variable is considered to be a literal name, and therefore does need the quotes.

attribute and map views

As defined in section 3.5, the AP allows users to query the dictionary to obtain useful information about the set of functions associated with a particular class. Such a set is known as a *view*. Figure 5 gives examples of how the <attr_view> and <map_view> statements can be used in an ADAMS program. Note that the view includes *all* associated attributes (or maps) by going up the inheritance tree. Also, this statement acts as an assignment in that the current set value is deleted before the attributes (or maps) that form the view are inserted.

commas

The AP parser allows commas to be inserted into any ADAMS statement, wherever the user wishes to use them to increase conceptual clarity. For example, an enumerated set may appear as "{ A B C}" or as "{ A, B, C }". Either is perfectly acceptable, and the two are treated as equivalent.

exiting for all loop

```
<<    <attr_set_class> is a SET of ATTRIBUTE elements        >>
<<    <attr_set_inst> belongs to <attr_set_class>        >>
<<    <attr_set_inst> view of <class_name>        >>
<<    for all attr in <attr_set_inst> do ....        >>

<<    <map_set_class> is a SET of MAP elements        >>
<<    <map_set_inst> belongs to <map_set_class>        >>
<<    <map_set_inst> map of <class_name>        >>
<<    for all attr in <map_set_inst> do ....        >>
```

Figure 5. Using Views

To exit a for all loop before every member of the set has been iterated over, use the following host language function call: break_for_loop("<loop_variable_name>"). Note that the loop variable name must be surrounded by quotations, since it is a literal name. Also, the loop is not exited immediately upon execution of the "break_for_loop" call. Rather, the entire loop body (if any statements remain after the break) is completed and then the loop is terminated.

5. Language Semantics

This section contains a detailed interpretation of the semantics of statements implemented by the AP. Readers who do not actually plan to use the AP may find this information too specific. Before reading this section, the reader should be familiar with the grammar defined in section 4 of this report.

This semantic definition characterizes a particular statement in two ways. First, the actions taken during statement execution are described. Each statement consists of individual components (or parameters), which are denoted by square brackets for emphasis. For example, the name of a new class, which is a component of the set class definition statement, is indicated by "<name>". Optional components are marked with square brackets (e.g., [<name>]). Second, typical run time errors are discussed. A statement is executed only if no such errors occur. Upon encountering an invalid statement, the AP displays an error message and the entire ADAMS program is terminated. Unless otherwise mentioned, the current AP implementation validates each of the error conditions listed.

The term "valid dictionary" appears in many of the error descriptions. If a <scope clause> is an explicit component of the statement, than a valid dictionary is one with a scope "greater than" or "equal to" the designated scope. See section 3.3 for a definition of "scope" and section 4.3 for an explanation of scope ordering). If no specific <scope clause> is given, then any dictionary may be valid (in this case all dictionary searches would begin with the default, or "lowest", scope).

Every dictionary entry must belong to one of these three types: CO_DOMAIN, CLASS, or INSTANCE. The CO_DOMAIN category contains only co-domain definitions and the CLASS category includes system, user defined, and set classes. Each CLASS entry is considered to have a *root,* which is the class in the root position of the tree structure, or class hierarchy, formed by inheritance. See section 3.2 for a definition of inheritance. An INSTANCE refers to an instance of any class.

5.1. Co-domain Definition

This statement defines a new co-domain with the following components: <name>, <regular expression>, [<store clause>], [<fetch clause>], [<scope clause>]. If successfully executed, a new CO_DOMAIN entry is made in the designated dictionary.

invalid name: If the <scope clause> specifies a permanent scope, then the <name> cannot already exist in the corresponding dictionary.

invalid conversion function:
Conversion functions identified by the <store clause> or the <fetch clause> must exist (i.e., be a valid C function) and perform the required operation correctly (i.e., receive the proper argument and return the expected value). NOTE: validation not implemented.

5.2. Class Definition

User Defined Class

This statement defines a new user defined class with the following components: <name>, <parent>, [<scope clause>]. If successfully executed, a new CLASS entry is made in the designated dictionary.

invalid name: If the <scope clause> specifies a permanent scope, then the <name> cannot already exist in the corresponding dictionary.

invalid parent: the <parent> must be an existing CLASS entry in a valid dictionary.

Set Class

This statement defines a new set class with the following components: <name>, <element type>, [<scope clause>]. If successfully executed, a new CLASS entry is made in the designated dictionary. Its parent is the SET system class and, thus, it is referred to as a SET CLASS.

invalid name: If the <scope clause> specifies a permanent scope, then the <name> cannot already exist in the corresponding dictionary.

invalid element type:
The <element type> must be an existing CLASS entry in a valid dictionary.

Association

This statement is actually a component in either a user defined or set class definition statement. It creates an association clause (for the <class> being defined) with the following components: [<synonym>], <set>. If successfully executed, adds a new association clause to the group of such clauses defined for the CLASS entry corresponding to <class>.

invalid synonym:
The <synonym> cannot already exist as a synonym for <class>. NOTE: validation not implemented.

invalid set: If a specific <set> is given, its root class must be an existing SET CLASS entry in a valid dictionary. The element type of the root must be either ATTRIBUTE or MAP.

invalid set enumeration:
If an enumerated <set> is given, each member must be an existing INSTANCE entry in a valid dictionary. All the enumerated instances must be of the same class (either ATTRIBUTE or MAP).

Restriction

This statement is actually a component of either a user defined or set class definition statement. It creates a class restriction clause (for the <class> being defined) with the following components: <attribute>, <relative op>, <const>. If successfully executed, adds a new restriction clause to the set of such clauses defined for the CLASS entry corresponding to <class>.

invalid attribute:
The <attribute> must be an existing INSTANCE entry (of class ATTRIBUTE) in a valid dictionary. It must also be an attribute associated with <class>.

5.3. Instance Creation

Attribute Class

This statement creates an instance of the ATTRIBUTE class with the following components: <name>, <image>, <value assign clause>, [<undefined clause>], [<scope clause>]. If successfully executed, a new INSTANCE entry is made in the designated dictionary.

invalid name: If the <scope clause> specifies a permanent scope, then the <name> cannot already exist in the corresponding dictionary.

invalid image: The <image> must be an existing CO_DOMAIN entry in a valid dictionary.

Map Class

This statement creates an instance of the MAP class with the following components: <name>, <image>, <value assign clause>, [<scope clause>]. If successfully executed, a new INSTANCE entry is made in the designated dictionary.

invalid name: If the <scope clause> specifies a permanent scope, then the <name> cannot already exist in the corresponding dictionary.

invalid image: The <image> must be an existing CLASS entry in a valid dictionary.

User Defined and Set Class

This statement creates an instance of a user defined or set class with the following components: <name>, <class>, [<set>], [<scope clause>]. If successfully executed, a new INSTANCE entry is made in the designated dictionary. All the attributes for the instance (i.e., all those associated with <class>) are initialized to their proper undefined values. If the instance is a set and a <set> value is given, then it is initialized accordingly.

invalid name: If the <scope clause> specifies a permanent scope, then the <name> cannot already exist in the corresponding dictionary.

invalid class: The <class> must be an existing CLASS entry in a valid dictionary.

invalid set: If a specific <set> is given, its root must be an existing SET CLASS entry in a valid dictionary. The root of <class> must also be an existing SET CLASS entry in a valid dictionary. The element types of both roots must be the same.

invalid set enumeration:
If an enumerated <set> is given, each member must be an existing INSTANCE entry in a valid dictionary. All the enumerated instances must be the same class. The root of <class> must be an existing SET CLASS entry in a valid dictionary. The element type of the root must be the same class as that of the enumerated instances.

5.4. Instance Manipulation

Attribute Assignment

This statement assigns the <value> to the given <attribute>. If successfully executed, it performs the assignment.

invalid attribute:
The <attribute> must evaluate to a valid attribute associated with an INSTANCE entry in a valid dictionary.

invalid value: The <value> must be a member of the co-domain defined for the <attribute>'s image. NOTE: validation not implemented.

Attribute Store

This statement assigns the <value> of a host language variable to the given <attribute>. If successfully executed, performs the assignment.

invalid attribute:
The <attribute> must evaluate to a valid attribute associated with an INSTANCE entry in a valid dictionary.

invalid value: The <value> must be a member of the co-domain defined for the <attribute>'s image. NOTE: validation not implemented.

Attribute Fetch

This statement assigns the <value> of the given attribute to a host language variable. If successfully executed, performs the assignment.

invalid attribute:
The <attribute> must evaluate to a valid attribute associated with an INSTANCE entry in a valid dictionary.

Map Assignment

This statement assigns the given <instance> as the new value of the <map>. If successfully executed, performs the assignment.

invalid map: The <map> must evaluate to a valid map associated with an INSTANCE entry in a valid dictionary.

invalid instance:
The <instance> must evaluate to an INSTANCE entry in a valid dictionary.

Instance Assignment

This statement assigns the values of all attributes and maps associated with the class of the given <target instance>. For set instances, the current set value is copied as well. The new values are obtained by copying those of the <value instance>.

invalid instance:
Both the <target instance> and the <value instance> must evaluate to INSTANCE entries in a valid dictionary. The root classes of the two instances must be the same or the root class of the <target instance> must be an ancestor in the class hierarchy of the <value instance>. In the latter case, only the functions associated with the <target instance> are copied.

Set Insertion

This statement inserts the designated <member> into the <set>. Note that the <member> instance is not actually copied (i.e., the insertion merely points to the existing instance entry). If successfully executed, the insertion is performed. Note that if the instance is already a member of the set, no error occurs.

invalid member:
The <member> must evaluate to an existing INSTANCE entry in a valid dictionary.

invalid set: The <set> must evaluate to an existing INSTANCE entry in a valid dictionary. The root of the <set>'s class must be an existing SET CLASS entry in a valid

dictionary.

invalid insert: The element type of the root of the <set>'s class must be the same as the class of the <member>.

Set Deletion

This statement deletes the designated <member> from the <set>. Note that the <member> is not removed from the dictionary, it is simply no longer a member of the <set>. If successfully executed, the deletion is performed.

invalid member:
The <member> must evaluate to an existing INSTANCE entry in a valid dictionary.

invalid set: The <set> must evaluate to an existing INSTANCE entry in a valid dictionary. The root of the <set>'s class must be an existing SET CLASS entry in a valid dictionary.

invalid delete: The <member> must be an existing member of <set>.

Set Iteration

This statement iterates over each member of the <set> by associating successive members with the given <variable>. If successfully executed, allows any statement in the loop body to reference the current member by referencing <variable>.

invalid set: The <set> must evaluate to an existing INSTANCE entry in a valid dictionary. The root of the <set>'s class must be an existing SET CLASS entry in a valid dictionary.

5.5. Dictionary Interrogation

See sections 3.5 and 4.3 of this report for a description of dictionary interrogation statements.

6. Examples

This section contains examples of ADAMS source files. Output from sample runs of some of the programs is also shown. These programs allow the user to create and manipulate entities in a university student/faculty database. The order in which the programs are compiled and executed makes a difference. Order is significant because of dependencies on previously created definitions and instances.

These source files are located on babbage in the directory "/va1/clk3h/gra/new_code/examples". An actual AP user may wish to copy them into his own directory and try running them in order to get a feel for the basic scenario. See section 2 of this paper for specific instructions on using the AP. Don't forget to run the "adams_init" utility first! If you use "adams_exec" and get an error "Cannot find include file conv_func.dict", then you didn't execute this initialization properly.

6.1. Defining the Persistent Database

```
#include "conv_func.dict"

main()
        /*
        ** This program creates a database structure based on the one described
        ** in the paper "ADAMS Interface Language" presented at the Hypercube
        ** Conference January 1988. The significant change is that students
        ** and faculty are subclasses of the class of PEOPLE, and consequently
        ** inherit all 'people' properties. The program does not actually
        ** insert any elements into the sets (i.e., relations) it creates.
        */
        {
        DICT_init();                                    /* necessary DB initialization */

                                                        /* co_domain definitions */
<<      string20     is a CO_DOMAIN
                consisting of #[a-zA-Z0-9]{1,20}#
                with scope USER  >>
<<      academic_rank  is a CO_DOMAIN
                consisting of
                #(research|visiting|)(full|associate|assistant)professor#
                with scope USER  >>
<<      dept_codes    is a CO_DOMAIN
                consisting of #[0-3][0-9]#
                with scope USER  >>
<<      course_nbrs   is a CO_DOMAIN
                consisting of #[A-Z]{2,4}[0-9]{3}#
                with scope USER  >>
<<      academic_terms is a CO_DOMAIN
                consisting of #[8-9][0-9][1-3]#
                with scope USER  >>
<<      S_S_nbrs      is a CO_DOMAIN
                consisting of #[0-9]{9}#
                with scope USER  >>
<<      grade_options  is a CO_DOMAIN
                consisting of
                #A+|A|A-|B+|B|B-|C+|C|C-|D+|D|D-|F|INC|P|WP|WF#
                with scope USER  >>
<<      date            is a CO_DOMAIN
                consisting of #[0-9]{2}/[0-9]{2}/88#
                with scope USER  >>
```

```
                                                    /* attribute definitions */
<<      name   belongs to ATTRIBUTE
                        with image string20, value is assigned
                        with scope USER   >>
<<      rank   belongs to ATTRIBUTE
                        with image dept_codes, value is assigned
                        with scope USER >>
<<      dept   belongs to ATTRIBUTE
                        with image dept_codes, value is assigned
                        with scope USER >>
<<      c_nbr  belongs to ATTRIBUTE
                        with image course_nbrs, value is assigned
                        with scope USER >>
<<      c_name   belongs to ATTRIBUTE
                        with image string20, value is assigned
                        with scope USER >>
<<      term   belongs to ATTRIBUTE
                        with image academic_terms, value is assigned
                        with scope USER >>
<<      major  belongs to ATTRIBUTE
                        with image dept_codes, value is assigned
                        with scope USER >>
<<      soc_sec_nbr  belongs to ATTRIBUTE
                        with image S_S_nbrs, value is assigned
                        with scope USER >>
<<      b_date    belongs to ATTRIBUTE
                        with image date, value is assigned,
                        with scope USER >>
<<      grade  belongs to ATTRIBUTE
                        with image grade_options, value is assigned
                        with scope USER >>
<<      date_last_mod belongs to ATTRIBUTE
                        with image date, value is assigned,
                        with scope USER >>
                                                    /* class declaration required for  */
                                                    /* the following map functions     */
<<      PERSON_REC is a CLASS
                        having data_fields = { name, soc_sec_nbr, b_date },
                        with scope USER >>
<<      FACULTY_REC is a PERSON_REC
                        having fac_data_fields = { rank, dept },
                        with scope USER >>
<<      FACULTY is a SET
                        of FACULTY_REC elements,
                        having { date_last_mod },
                        with scope USER >>
                                                    /* map functions */
<<      advisor   belongs to MAP
                        with image FACULTY_REC, value is assigned,
                        with scope USER >>
<<      instructor belongs to MAP
                        with image FACULTY_REC, value is assigned,
                        with scope USER >>
                                                    /* class declarations required   */
                                                    /* for the following map functions */
<<      STUDENT_REC is a PERSON_REC
                        having stu_data_fields = { major },
                        having maps = { advisor },
                        with scope USER >>
<<      STUDENTS   is a SET
```

```
                        of STUDENT_REC elements,
                        with scope USER >>
<<      COURSE_REC is a CLASS
                        having data_fields = { c_nbr, c_name, term },
                        having maps = { instructor },
                        with scope USER >>
<<      COURSES    is a SET
                        of COURSE_REC elements,
                        with scope USER >>
                                                /* map functions for many-to many */
                                                /* enrollment relationship       */
<<      student belongs to MAP
                        with image STUDENT_REC, value is assigned,
                        with scope USER >>
<<      course  belongs to MAP
                        with image COURSE_REC, value is assigned,
                        with scope USER >>
<<      ENROLL_REC is a CLASS
                        having data_fields = { grade },
                        having maps = { student, course },
                        with scope USER >>
<<      ENROLLMENT is a SET
                        of ENROLL_REC elements,
                        with scope USER >>
                                                /* FINALLY, the 6 actual data sets */
<<      courses      belongs to COURSES, with scope USER      >>
<<      enrollment   belongs to ENROLLMENT, with scope USER   >>
<<      tenured      belongs to FACULTY, with scope USER      >>
<<      untenured    belongs to FACULTY, with scope USER      >>
<<      graduate     belongs to STUDENTS, with scope USER     >>
<<      undergrad    belongs to STUDENTS, with scope USER     >>

        DICT_close();
        }
```

6.2. Entering Data

6.2.1. Source File

```
#include <stdio.h>
#include "conv_func.dict"

#define TRUE 1
#define FALSE 0

main()
        /*
        ** This program provides a simple-minded data entry capability
        ** for the elements in the basic sets of the "school database"
        ** described in the "ADAMS Interface Language" paper presented
        ** at the Hypercube Conference, Jan 1988.
        **
        ** It allows the user to select any of the six basic data sets
        **       tenured (FACULTY)
        **       untenured (FACULTY)
        **       undergrad (STUDENTS)
        **       graduate (STUDENTS)
        **       courses (COURSES)
        **       enrollment (ENROLLMENT)
        ** and to create instances with the associated attribute and
```

```
        ** map values.
        */
        {
        char    response[20], attr_value[30], query_value[30];
        int     found, running;

        DICT_init();

<<      name_variable     rec, set_inst, fac_type                    >>

<<      fac_rec   belongs to FACULTY_REC >>                  /* LOCAL ADAMS instances */
<<      stu_rec   belongs to STUDENT_REC >>
<<      course_rec belongs to COURSE_REC >>
<<      enroll_rec belongs to ENROLL_REC >>

        printf ("The 'school database' has 6 data sets (or relations). They are:\n");
        printf ("\ttenured (FACULTY) \n\tuntenured (FACULTY)\n");
        printf ("\tgraduate (STUDENTS) \n\tundergrad (STUDENTS)\n");
        printf ("\tcourses (COURSES) \n\tenrollment (ENROLLMENT)\n");
        running = TRUE;
        while (running)
                {
                printf ("\nEnter name of data set to accept entry ('q' to quit) > ");
                scanf ("%s", response);
                switch (response[0])
                 {
                 case 't':                                   /* 'tenured' faculty input */
                        enter_faculty("fac_rec");
<<                      insert fac_rec into tenured >>
                        break;
                 case 'u':
                        switch (response[2])
                         {
                         case 't':                           /* 'unTenured' faculty input */
                                enter_faculty("fac_rec");
<<                              insert fac_rec into untenured >>
                                break;
                         case 'd':                           /* 'unDergrad' input */
                                enter_student("stu_rec");
                                                            /* find student's advisor */
                                printf ("When correct advisor appears respond with 'yes', else 'no'\n");
                                found = find_faculty("advisor", "tenured","stu_rec");
                                if (!found)                 /* search untenured faculty */
                                        find_faculty("advisor", "untenured","stu_rec");
<<                              insert stu_rec into undergrad >>
                                break;
                         default:
                                printf ("\t%s is an unrecognized response\n", response);
                                printf ("\t\tEnter data set name (in lower case) or\n");
                                printf ("\t\t'q' to exit the program.\n");
                                break;
                         }
                        break;
                 case 'g':                                   /* 'graduate' input */
                        enter_student("stu_rec");
                                                            /* find student's advisor    */
                                                            /* Note--only tenured faculty */
                                                            /* may advise grad students  */
                        printf ("When correct advisor appears respond with 'yes', else 'no'\n");
                        find_faculty("advisor", "tenured", "stu_rec");
```

```
<<                      insert stu_rec into graduate >>
                        break;
                case 'c':                                   /* 'courses' input */
                        enter_course("course_rec");
                                                            /* find course's instructor */
                        printf ("When correct instructor appears respond with 'yes', else 'no'\n");
                        found = find_faculty("instructor", "tenured", "course_rec");
                        if (!found)                         /* search untenured faculty */
                             find_faculty("instructor", "untenured", "course_rec");
<<                      insert course_rec into courses >>
                        break;
                case 'e':                                   /* 'enrollment' input */
                        enter_enroll("enroll_rec");
<<                      insert enroll_rec into enrollment >>
                        break;
                case 'q':
                        running = FALSE;
                        break;
                default:
                        printf ("\t%s is an unrecognized response\n", response);
                        printf ("\t\tEnter data set name (in lower case) or\n");
                        printf ("\t\t'q' to exit the program\n");
                        break;
                }
                }

        DICT_close();
        }

enter_faculty(rec)
char *rec;
        /*
        ** This function allows the user to create a new instance of the 'FACULTY_REC' class.
        */
        {
        char attr_value[30];

        printf ("Enter faculty name > ");
        scanf ("%s", attr_value);
<<      store into rec.name from attr_value >>
        printf ("Enter soc.sec. nbr > ");
        scanf ("%s", attr_value);
<<      store into rec.soc_sec_nbr from attr_value >>
        printf ("Enter birth date  > ");
        scanf ("%s", attr_value);
<<      store into rec.b_date from attr_value >>
        printf ("Enter current rank > ");
        scanf ("%s", attr_value);
<<      store into rec.rank from attr_value >>
        printf ("Enter department  > ");
        scanf ("%s", attr_value);
<<      store into rec.dept from attr_value >>
        }

enter_student(rec)
char *rec;
        /*
        ** This function allows the user to create a new instance of the 'STUDENT_REC' class.
        */
        {
```

```
        char attr_value[30];

        printf ("Enter student name   > ");
        scanf  ("%s", attr_value);
<<      store into rec.name from attr_value >>
        printf ("Enter soc.sec. nbr > ");
        scanf ("%s", attr_value);
<<      store into rec.soc_sec_nbr from attr_value >>
        printf ("Enter birth date   > ");
        scanf ("%s", attr_value);
<<      store into rec.b_date from attr_value >>
        printf ("Enter student's major > ");
        scanf  ("%s", attr_value);
<<      store into rec.major from attr_value >>
        }

find_faculty(fac_type, set_inst, rec)
char *fac_type, *set_inst, *rec;
        /*
        ** This function searches the given set instance to find the desired faculty member.
        */
        {
        int found;
        char attr_value[30], response[30];

        found = FALSE;
<<      for all f in set_inst do
<<              fetch into attr_value from f.name >>
                printf ("\t\t%s, ", attr_value);
<<              fetch into attr_value from f.dept >>
                printf ("%s  ? ", attr_value);
                scanf ("%s", response);
                if (*response == 'y')
                        {
                        found = TRUE;
<<                      assign into rec.fac_type from f >>
                        break_for_loop("f");
                        }
        >>
        return(found);
        }

find_student(name, set_inst, rec)
char *name, *set_inst, *rec;
        /*
        ** This function searches the given set instance to find the desired student.
        */
        {
        int found;
        char attr_value[30], response[30];

        found = FALSE;
<<      for all s in set_inst do
<<              fetch into attr_value from s.name >>
                if (strcmp(name, attr_value) == 0)
                        {
                        found = TRUE;
<<                      assign into rec.student from s >>
                        break_for_loop("s");
                        }
```

```
        >>
        return(found);
        }

enter_course(rec)
char *rec;
        /*
        ** This function allows the user to create a new instance of the 'COURSE_REC' class.
        */
        {
        char attr_value[30];

        printf ("Enter course number XXnnn  > ");
        scanf ("%s", attr_value);
<<      store into rec.c_nbr from attr_value >>
        printf ("Enter course name        > ");
        scanf ("%s", attr_value);
<<      store into rec.c_name from attr_value >>
        printf ("Enter course term e.g. f88 > ");
        scanf ("%s", attr_value);
<<      store into rec.term from attr_value >>
        }

enter_enroll(rec)
char *rec;
        /*
        ** This function allows the user to create a new instance of the 'ENROLL_REC' class.
        */
        {
        int found;
        char attr_value[30], response[30], query_value[30];

        printf ("Enter student name > ");
        scanf ("%s", query_value);
        printf ("\tIs the student undergraduate ('y' or 'n')? ");
        scanf ("%s", response);
        if (*response == 'y')                                   /* search 'undergraduate' */
                found = find_student(query_value, "undergrad", rec);
        else                                                    /* search 'graduate' */
                found = find_student(query_value, "graduate", rec);
        if (!found)
                {
                printf ("\tstudent - %s - unknown, entry aborted\n", query_value);
                return(0);
                }
        printf ("Enter course number > ");
        scanf ("%s", query_value);
        found = FALSE;
<<      for all c in courses do
<<              fetch into attr_value from c.c_nbr >>
                if (strcmp(query_value, attr_value) == 0)
                        {
                        found = TRUE;
<<                      assign into rec.course from c >>
                        break_for_loop("c");
                        }
        >>
        if (!found)
                {
                printf ("\tcourse - %s - unknown, entry aborted\n", query_value);
```

```
                return(0);
                }
        printf ("Enter grade received > ");
        scanf ("%s", attr_value);
<<      store into rec.grade from attr_value >>
        }
```

6.2.2. Sample Output

```
The 'school database' has 6 data sets (or relations). They are:
        tenured (FACULTY)
        untenured (FACULTY)
        graduate (STUDENTS)
        undergrad (STUDENTS)
        courses (COURSES)
        enrollment (ENROLLMENT)

Enter name of data set to accept entry ('q' to quit) > enrollment

Enter student name > Smith
        Is the student undergraduate ('y' or 'n')? y
Enter course number > CS655
Enter grade received > B+

Enter name of data set to accept entry ('q' to quit) > graduate

Enter student name   > Thomas
Enter soc.sec. nbr > 111-22-3333
Enter birth date   > 5/1/60
Enter student's major > CS
When correct advisor appears respond with 'yes', else 'no'
                Pfaltz, CS  ? no
                Reynolds, CS  ? yes

Enter name of data set to accept entry ('q' to quit) > q
```

6.3. Interrogating the Dictionary

6.3.1. Source File

```
#include <stdio.h>
#include "conv_func.dict"

#define TRUE 1
#define FALSE 0

main()
        /*
        ** This program supports an interactive interface which inputs names
        ** of elements in the permanant name space and then describes the
        ** indicated instance or class.
        */
        {
        char    name[30];
        int     running;

        DICT_init();

        running = TRUE;
        while (running)
```

```
        in_loop = FALSE;
<<      for all x in attr_view do
                in_loop = TRUE;
                indent (depth);
                printf ("\t\t%s\n", dict_name_of ("x") );
                >>
        if (!in_loop)
                printf("\t\tnone\n");

<<      map_view map of the_class                       >>
        indent (depth);
        printf ("\tList of Associated Maps:\n");
        in_loop = FALSE;
<<      for all x in map_view do
                in_loop = TRUE;
                indent (depth);
                printf ("\t\t%s -> %s\n", dict_name_of ("x"), image_of("x") );
                >>
        if (!in_loop)
                printf("\t\tnone\n");
        }

indent (depth)
int     depth;
        /*
        ** Indent a printed line by "depth" tabs
        */
        {
        int     i;

        for (i=0; i<depth; ++i)
                printf ("\t");
        }
```

6.3.2. Sample Output

```
Enter class or instance name ('q' to quit) > enrollment

enrollment belongs to class ENROLLMENT with the following properties
        List of Associated Attributes:
                none
        List of Associated Maps:
                none
        SET instance composed of ENROLL_REC elements
                                List of Associated Attributes:
                                        grade
                                List of Associated Maps:
                                        student -> STUDENT_REC
                                        course -> COURSE_REC

Enter class or instance name ('q' to quit) > graduate

graduate belongs to class STUDENTS with the following properties
        List of Associated Attributes:
                none
        List of Associated Maps:
                none
        SET instance composed of STUDENT_REC elements
                                List of Associated Attributes:
                                        major
```

```
                                        name
                                        soc_sec_nbr
                                        b_date
                                List of Associated Maps:
                                        advisor -> FACULTY_REC

Enter class or instance name ('q' to quit) > FACULTY_REC

FACULTY_REC is a class with the following properties
        List of Associated Attributes:
                rank
                dept
                name
                soc_sec_nbr
                b_date
        List of Associated Maps:
                none

Enter class or instance name ('q' to quit) > courses
courses belongs to class COURSES with the following properties
        List of Associated Attributes:
                none
        List of Associated Maps:
                none
        SET instance composed of COURSE_REC elements
                                List of Associated Attributes:
                                        c_nbr
                                        c_name
                                        term
                                List of Associated Maps:
                                        instructor -> FACULTY_REC

Enter class or instance name ('q' to quit) > q
```

6.4. Displaying Data

6.4.1. Source File

```
#include <stdio.h>
#include "conv_func.dict"

#define TRUE 1
#define FALSE 0

main()
        /*
        ** This program provides a simple-minded display capability for the elements
        ** in the basic sets (i.e., relations) of the "school database" described
        ** in the "ADAMS Interface Language" paper presented at the Hypercube
        ** Conference, January 1988.
        **
        ** It allows the user to select any of the six basic data sets:
        **      tenured (FACULTY)
        **      untenured (FACULTY)
        **      undergrad (STUDENTS)
        **      graduate (STUDENTS)
        **      courses (COURSES)
        **      enrollment (ENROLLMENT)
        **
        ** The program then displays the attribute values of the current members
```

```
        ** of the selected data set.
        */
        {
        char    attr_value[30], element_class[30], query_value[30], response[20];
        int     running;

        DICT_init();

<<      name_variable     element_class                 >>
<<      SCHEMA is a SET of ATTRIBUTE elements   >>
<<      attr_set belongs to SCHEMA            >>

        printf ("The 'school database' has 6 data sets (or relations). They are:\n");
        printf ("\ttenured (FACULTY) \n\tuntenured (FACULTY)\n");
        printf ("\tgraduate (STUDENTS) \n\tundergrad (STUDENTS)\n");
        printf ("\tcourses (COURSES) \n\tenrollment (ENROLLMENT)\n");
        running = TRUE;
        while (running)
                {
                printf ("\nEnter name of data set to be displayed ('q' to quit) > ");
                scanf ("%s", response);
                printf("\n");
                switch (response[0])
                 {
                 case 't':
                        printf ("\nDisplay of 'tenured' faculty\n");
                        strcpy (element_class, class_of_element("tenured") );
<<                      attr_set view of element_class          >>
<<                      for all x in tenured do
                                display_atts ("x", "attr_set");
                                printf ("\n");
                        >>
                        printf ("\n");
                        break;
                 case 'u':
                        switch (response[2])
                         {
                         case 't':
                                printf ("\nDisplay of 'untenured' faculty\n");
                                strcpy (element_class, class_of_element("untenured") );
<<                              attr_set view of element_class          >>
<<                              for all x in untenured do
                                        display_atts ("x", "attr_set");
                                        printf ("\n");
                                >>
                                printf ("\n");
                                break;
                         case 'd':
                                printf ("\nDisplay of 'undergraduate' students\n");
                                strcpy (element_class, class_of_element("undergrad") );
<<                              attr_set view of element_class          >>
<<                              for all x in undergrad do
                                        display_atts ("x", "attr_set");
<<                                      fetch into attr_value from x.advisor.name >>
                                        printf ("advisor is    - %s\n", attr_value);
                                        printf ("\n");
                                >>
                                printf ("\n");
                                break;
                         default:
```

```
                        printf ("\t%s is an unrecognized response\n", response);
                        printf ("\t\tEnter data set name---in lower case\n");
                        printf ("\t\tor 'quit' to exit the program\n");
                        break;
                }
                break;
        case 'g':
                printf ("\nDisplay of 'graduate' students\n");
                strcpy (element_class, class_of_element("graduate") );
<<              attr_set view of element_class          >>
<<              for all x in graduate do
                        display_atts ("x", "attr_set");
<<                      fetch into attr_value from x.advisor.name >>
                        printf ("advisor is    - %s\n", attr_value);
                        printf ("\n");
                >>
                printf ("\n");
                break;
        case 'c':
                printf ("\nDisplay of 'courses' offered\n");
                strcpy (element_class, class_of_element("courses") );
<<              attr_set view of element_class          >>
<<              for all x in courses do
                        display_atts ("x", "attr_set");
<<                      fetch into attr_value from x.instructor.name >>
                        printf ("instructor is  - %s", attr_value);
<<                      fetch into attr_value from x.instructor.dept >>
                        printf (", %s\n", attr_value);
                        printf ("\n");
                >>
                printf ("\n");
                break;
        case 'e':
                printf ("\nDisplay of 'enrollment' by class\n");
<<              for all c in courses do
<<                      fetch into query_value from c.c_nbr >>
                        printf ("\t%s -", query_value);

<<                      for all e in enrollment do
<<                              fetch into attr_value from e.course.c_nbr >>
                                if (strcmp(attr_value, query_value) == 0)
                                        {       /* Display this enrollment */
<<                                      fetch into attr_value from e.student.name >>
                                        printf ("\t%s", attr_value);
<<                                      fetch into attr_value from e.grade >>
                                        printf ("\t- %s\n\t", attr_value);
                                        }
                                >>
                        printf ("\n");
                        >>
                break;
        case 'q':
                running = FALSE;
                break;
        default:
                printf ("\t%s is an unrecognized response\n", response);
                printf ("\t\tEnter data set name (in lower case) or\n");
                printf ("\t\t'q' to exit the program.\n");
                break;
        }
```

```
                }
        DICT_close();
        }

display_atts (x, attr_set)
char    *x, *attr_set;
        /*
        ** Given an ADAMS instance with the indicated set of associated attributes,
        ** access each attribute value and display them one per line.
        */
        {
        char    attr_value[30];

<<      for all attr in attr_set do
<<              fetch into attr_value from x.attr >>
                printf ("%-15s- %s\n", dict_name_of("attr"), attr_value);
        >>
        }
```

6.4.2. Sample Output

The 'school database' has 6 data sets (or relations). They are:

tenured (FACULTY)
untenured (FACULTY)
graduate (STUDENTS)
undergrad (STUDENTS)
courses (COURSES)
enrollment (ENROLLMENT)

Enter name of data set to be displayed ('q' to quit) > **tenured**

Display of 'tenured' faculty
rank - prof.
dept - CS
name - Pfaltz
soc_sec_nbr - 001-00-0007
b_date - 6/25/35

rank - assoc.prof.
dept - CS
name - Reynolds
soc_sec_nbr - 002-00-1111
b_date - 4/23/45

Enter name of data set to be displayed ('q' to quit) > **graduate**

Display of 'graduate' students
major - CS
name - Klumpp
soc_sec_nbr - 123-45-6789
b_date - 8/13/53
advisor is - Reynolds

major - CS
name - Baron
soc_sec_nbr - 234-56-7890
b_date - 11/3/52
advisor is - Pfaltz

major - CS

```
name         - Orlandic
soc_sec_nbr   - 987-65-4321
b_date       - 2/12/51
advisor is    - Pfaltz

major        - CS
name         - Thomas
soc_sec_nbr   - 1111-22-3333
b_date       - 5/1/60
advisor is    - Reynolds

Enter name of data set to be displayed ('q' to quit) > enrollment

Display of 'enrollment' by class
          CS662 -  Klumpp  - B+
                   Baron   - A

          CS655 -  Klumpp  - A
                   Baron   - B
                   Orlandic - A-
                   Smith   - B+

Enter name of data set to be displayed ('q' to quit) > courses

Display of 'courses' offered
c_nbr        - CS662
c_name        - Database
term         - s88
instructor is  - Son, CS

c_nbr        - CS655
c_name        - Languages
term         - f88
instructor is  - Reynolds, CS

Enter name of data set to be displayed ('q' to quit) > q
```

6.5. Implementing Set Operators

6.5.1. Source File

```
#include <stdio.h>
#include "conv_func.dict"

main()
          /*
          ** This program tests the implementation of the hierarchical union
          ** (e.g., "h_union") facility.
          **
          ** The names of two operand sets are entered by the user. The
          ** current extent of these two sets is displayed and the name
          ** of the resulting set is requested. Finally, the union is formed
          ** and the extent of the resultant set is diplayed.
          */
          {
          char       operand1[20], operand2[20], result[20];

          DICT_init();

<<        name_variable       X, Y, Z, x_class, y_class, CLASS_z, CLASS_Z,
```

```
                        z_elem, inst, element_class >>

        accept_set_name (operand1);
        show_set (operand1);
        accept_set_name (operand2);
        show_set (operand2);
        accept_new_name (result);
        h_union (operand1, operand2, result);
        show_set (result);

        DICT_close();
        }

a_union (X, Y, Z)
char    *X, *Y, *Z;
        /*
        ** Form          Z <- X union Y
        ** where we assume that
        **  1.    the instance 'Z' exists, and
        **  2. 'Z' is empty.
        **
        ** NOTE: This is easily implemented at the system level as
        **          an O(n) process rather than the O(n^2) process here.
        */
        {

        if (strcmp(class_of_element(X), class_of_element(Y)) != 0
                || class_of_element(X) == '\0')
                {                                       /* X and Y are not conformable      */
                                                        /* or X is not a set instance   */
                return (0);
                }

<<      assign into Z from X            >>              /*  Z <- X  */
<<      for all y in Y do
                if (!is_element_of("y", X) )  /* don't duplicate elements */
<<                      insert y into Z                 >>
        >>
        return (1);
        }

a_intersect (X, Y, Z)
char    *X, *Y, *Z;
        /*
        ** Form          Z <- X intersect Y
        ** where we assume that
        **  1.    the instance 'Z' exists, and
        **  2. 'Z' is empty.
        **
        ** NOTE: This is easily implemented at the system level as
        **          an O(n) process rather than the O(n^2) process here.
        */
        {
        if (strcmp(class_of_element(X), class_of_element(Y)) != 0
                || class_of_element(X) == '\0')
                {                                       /* X and Y are not conformable      */
                                                        /* or X is not a set instance   */
                return (0);
                }
```

```
<<      for all y in Y do
                if (is_element_of("y", X) )
<<                      insert y into Z                 >>
        >>
        return (1);
        }

h_union (X, Y, Z)
char    *X, *Y, *Z;
        /*
        ** This procedure creates the set 'Z' <- 'X' union 'Y'
        ** in an environment that supports class hierarchies.
        ** (In this environment we do not require X and Y to consist
        **  of elements of the same class; instead 'CLASS_z' will consist
        **  of those attributes which are common to both.)
        **
        ** NOTE: In this version, if the two operand sets are not of the
        **       same class,  'CLASS_z' and 'CLASS_Z' are established
        **       within the procedure, and the instance 'Z' created.
        **       One could as well expect the invoking procedure to
        **       create the set instance 'Z' and thus its class.
        **
        ** NOTE: This version determines the common attributes.
        **       An alternate approach would search the dictionary for
        **       the greatest common parent (if any) and use that as the
        **       class of the result.
        */
        {
        char    x_class[20], y_class[20];
        char    z_elem[20], CLASS_z[20], CLASS_Z[20];

        strcpy (x_class, class_of_element(X) );         /* get operand classes */
        strcpy (y_class, class_of_element(Y) );

        if (strcmp (x_class, y_class) == 0)
                {                                       /* EASY case, sets are of the same class*/
                strcpy (CLASS_Z, class_of(X));
<<              Z       belongs to CLASS_Z                                      >>
                a_union (X, Y, Z);
                return (1);
                }
                                                        /* HARD case, sets are not conformable */
                                                        /* find their common attributes     */
<<      x_attr_view     belongs to ATTRIBUTES                                   >>
<<      y_attr_view     belongs to ATTRIBUTES                                   >>
<<      x_attr_view     view of x_class                                         >>
<<      y_attr_view     view of y_class                                         >>
<<      result_attrs    belongs to ATTRIBUTES                                   >>
        a_intersect ("x_attr_view", "y_attr_view", "result_attrs");

        make_class_name (CLASS_z);
<<      CLASS_z                 is a CLASS, having result_attrs                 >>
        make_class_name (CLASS_Z);
<<      CLASS_Z                 is a SET, of CLASS_z elements                           >>
<<      Z               belongs to CLASS_Z                                      >>

<<      for all x in X do
                make_name (z_elem);             /* generate arbitrary name */
<<              z_elem belongs to CLASS_z                               >>
<<              for all attr in result_attrs do
```

```
<<                      assign into z_elem.attr from x.attr                 >>
            >>
<<          insert z_elem into Z                                                    >>
        >>

<<      for all y in Y do
            if (!is_element_of("y", X) )
                    {                           /* don't include duplicate elements */
                    make_name (z_elem);         /* generate arbitrary name */
<<                  z_elem belongs to CLASS_z                           >>
<<                  for all attr in result_attrs do
<<                          assign into z_elem.attr from y.attr         >>
                    >>
<<                  insert z_elem into Z                                        >>
                    }
        >>
        return (1);
        }

int     n_count = 0;

make_class_name (name)
char    *name;
        /*
        ** Generate a unique LOCAL CLASS name
        */
        {
        sprintf (name, "%s_%d", "T_CLASS", ++n_count);
        }

make_name (name)
char    *name;
        /*
        ** Generate a unique LOCAL name
        */
        {
        sprintf (name, "%s_%d", "TEMP", ++n_count);
        }

show_set (inst)
char    *inst;
        /*
        ** Display the attributes of elements in the named set instance.
        ** This is a generic, but not particularly elegant procedure.
        */
        {
        char    attr_value[30], element_class[30];

        printf ("Display of the set '%s'\n", inst);
<<      attr_set belongs to ATTRIBUTES                                      >>
        strcpy (element_class, class_of_element(inst) );
<<      attr_set view of element_class                                      >>
<<      for all x in inst do
<<          for all attr in attr_set do
<<              fetch into attr_value from x.attr       >>
                printf ("%-12s ", attr_value);
                >>
            printf ("\n");
            >>
        printf ("\n");
```

```
        }

accept_set_name (set_name)
char    *set_name;
        {
        int         accepted;

        accepted = 0;
        while (!accepted)
                {
                printf ("Enter the name of a set instance > ");
                scanf ("%s", set_name);
                printf("\n");
                if (class_of_element(set_name) != 0)
                        accepted = 1;
                else
                        printf ("\t\t%s is NOT a set instance\n\n", set_name);
                }
        }

accept_new_name (name)
char    *name;
        {
        int         accepted;

        accepted = 0;
        while (!accepted)
                {
                printf ("Enter a new name to denote the result > ");
                scanf ("%s", name);
                printf("\n");
                if (is_dict_entry(name) )
                        printf ("\t\t%s already exists as a ADAMS name\n\n", name);
                else
                        accepted = 1;
                }
        }
```

6.5.2. Sample Output

Enter the name of a set instance > **tenured**

Display of the set 'tenured'

```
prof.       CS   Pfaltz     001-00-0007   6/25/35
assoc.prof. CS   Reynolds   002-00-1111   4/23/45
```

Enter the name of a set instance > **graduate**

Display of the set 'graduate'

```
CS   Klumpp    123-45-5678   8/13/53
CS   Baron     234-56-7890   11/3/52
CS   Orlandic  987-65-4321   2/12/51
CS   Thomas    111-22-3333   5/2/60
```

Enter a new name to denote the result > **people**

Display of the set 'people'

123-45-5678	8/13/53	Klumpp
001-00-0007	6/25/35	Pfaltz
002-00-1111	4/23/45	Reynolds
234-56-7890	11/3/52	Baron
987-65-4321	2/12/51	Orlandic
111-22-3333	5/2/60	Thomas

7. Conclusion

This section of the report summarizes the results of the AP project. First, the modifications to the ADAMS language which have already been incorporated into the current AP implementation are discussed. Most of these ideas will be integrated into the definition of the actual ADAMS interface language. However, the implementation and syntactic details may, of course, be changed after more careful consideration. Second, this conclusion lists modifications which have been proposed as a direct result of experimentation with writing ADAMS programs using the AP. Finally, the areas in which considerable design effort is still needed are identified. These are issues which are not thoroughly addressed in the current ADAMS language definition document [Pf87] and should serve as a check list for further design work.

7.1. Implemented Modifications

The following features, which were not defined in the original ADAMS design document, have been added to the AP system. For the most part, these features increase the data manipulation capabilities available to the user. The data definition facilities had been focused on in the initial design efforts and were therefore more extensively and better defined.

(1) The set of all attributes or maps associated with a given class may be obtained through a view statement (see section 3.5 and 4.3).

(2) For programming convenience, literal dictionary names in any ADAMS statement may be replaced by a dictionary name variable (see section 4.3).

(3) A small group of host language functions are supported by the AP version of ADAMS to allow user interrogation of dictionaries (see sections 3.5 and 4.2.6).

(4) An ADAMS "for all" loop may be exited before all members of the set have been accessed (see section 4.3).

(5) Data manipulation statements (including set insertion and assignment of attributes, maps, sets, and instances) have been more precisely defined (see sections 3.4 and 5.4).

7.2. Proposed Modifications

(1) Currently attributes and maps are treated as instances of the system classes ATTRIBUTE and MAP. However, it seems conceptually clearer to require that the user define a new class whose parent is one of these two classes, and then create instances of the new class. This allows the image of the function to be declared in the definition statement rather than in the instance creation statement. Two important issues become more apparent with this change. First, the map or attribute classes may now have their own attributes or maps associated them (as with all class definitions). The problem is that an ambiguity is introduced in trying to reference the attribute or map values of an instance whose class is itself a function. Second, attributes or maps are often collected together in a set. Under the current implementation, such a set is described as containing "ATTRIBUTE elements". However, the proposed change would require such a set to consist of instances which may have different parent classes (such as INTEGER_ATTRIBUTE, or DATE_ATTRIBUTE) but all have a common ancestor, the system defined ATTRIBUTE class. We must ensure that such a set is legal and that operations on it are well defined. The first issue is syntactic, and requires a revision of the basic ADAMS syntax. The second affects the semantics of set operations where one has a class hierarchy.

(2) The small set of dictionary interrogation facilities now supported must be greatly expanded. For example, routine information such as whether or not a name respresents a co-domain entry in the dictionary, what the current scope of an instance is, or whether a set

is empty should be available to the user. In addition, given two classes, the system should be able to return the "highest" common parent class (if a common class exists). There are numerous such interrogations that must be defined and included in the ADAMS definition.

(3) Because ADAMS requires all instances to have a unique name, it is desirable for the system to provide an automatic name generation routine. Often a user does not care what name is used and does not want to be bothered with the "house keeping" details necessary to ensure uniqueness. Probably this facility will differentiate between names for permanent (user scope) and temporary (local scope) instances.

(4) Although this is an implementation concern, it is worth noting here. The current AP system stores both the definitions and data (i.e., instances) in the dictionary. In an actual ADAMS implementation this would clearly not be the case. The instance names, of course, be recorded in the dictionary. However, the collection of data associated with each instance (i.e., the function values) will be stored separately, possibly in a different manner.

(5) This is a small syntactic observation, but will make ADAMS code a lot easier to write, debug, and read. All literal dictionary names should appear in quotations (as is the common convention in most programming languages). This change will then make the use of a name variable immediately apparent (currently the program must be examined to determine if a name is to be interpreted as a literal or a name variable), and eliminate the need for the adhoc name_variable ADAMS statement.

(6) The current implementation has included a "kludge" which allows temporary instances to be inserted into permanent sets. The system automatically converts the temporary instance into a permanent one before the insertion is made. The name of the new persistent instance is automatically chosen by the system (since it is not considered relevant to the user, who simply wants a collection of unnamed objects). Although such a mechanism is useful, it confuses several different issues (unnamed instances, rescoping, validity of operations that involve instances of different scopes, etc.). A careful reworking of these concepts is required (also see item 3 in this list). The entire issue of "copy" verses "reference" semantics needs to be systematically reviewed, and then reflected in the syntax (see the next paragraph as well).

(7) The semantics of the assignment statement have been questioned. The current implementation treats the assign statement as three separate actions. The desired interpretation is determined by the types of the arguments. If the left hand side (LHS) argument is an attribute designator then the assignment is interpreted as "attribute assignment", in which the co-domain value of the RHS is copied to become the new value of the LHS attribute. If the LHS argument is a map designator then the assignment is interpreted as "map assignment", in which the map is made to point to the instance designated by the RHS. Finally, if the LHS is a the name of an instance, then all the function values associated with the RHS instance are copied into the corresponding values of the LHS. One issue that arises is whether or not the assignment of instances should be handled as a separate copy statement. Such a statement would be used only when a user wishes to make such a copy. The advantages are two-fold. First, the change is desirable to clarify for the user which action is being carried out as a result of the statement execution. Second, the LHS instance could be designated by a map evaluation in addition to a explicit instance name. Currently such a designation is impossible because of the ambiguity as to whether the map value itself is being updated or the instance which is the current value of the map is to be modified.

An interesting outcome of the initial experimentation with the AP is a clearer understanding of ADAMS as a "minimalist" approach to building a database system. As stated in the introduction of this report, ADAMS is not intended to be an all-encompassing DBMS (with interactive user support, etc.). However, the basic primitives supported by ADAMS are intended to support the development of such a system. To aid the user (i.e., a programmer) in this job, it has been proposed that ADAMS provide a group of higher level uilities, built only upon the ADAMS primitives.

The method in which one would use ADAMS to model multi-valued attributes is used as an example of this approach. In ADAMS, multi-valued attributes are represented by maps which have a set class as their image. The set must then contain intances of a class which has the appropriate attribute associated with it. Figure 6 first gives the abbreviated version of a multi-valued attribute definition and then gives the actual ADAMS code needed to carry out the definition. Note that literal names are surrounded by quotations (as proposed) and functions are first defined as classes and then instances are created (as proposed above). The goal would be to support many such abbreviated statements (and let the system take care of generating the details). Facilities which simplify the definition of relational databases are also envisioned.

7.3. Unimplemented Features and Unresolved Issues

(1) syntax and use of predicates in restriction clauses (including how they will be used to enforce integrity constraints)

Abbreviated Version

```
<< "children" is a multi-valued instance of the "child" ATTRIBUTE
   having image "child_name"
>>
```

Expanded Code

```
<< X is an ATTRIBUTE with image "child_name"          >>
<< "child" belongs to X                               >>
<< Y is a CLASS having attribute = { "child" }        >>
<< Y is a SET of X elements                           >>
<< Z is a MAP with image Y                            >>
<< "children" belongs to Z                            >>
```

Accessing an Attribute Value

```
<< for all x in "joe"."children"
        << fetch into name from x."child"             >>
>>
```

Figure 6. Sample of ADAMS User-Friendly Utility

---

(2) error recovery/transaction mechanism

(3) partition/subset facility (including update propagation issues)

(4) "forward" definition statements (how they can be used, how to handle validation in respect to their use)

(5) system supported set operations: which ones are supported and how are they defined (of particular interest is how inheritance is dealt with in defining them)?

(6) parallelism: how is it incorporated into ADAMS (internal and external)?

(7) computed attribute values: when and how are the computations triggered and how are the computations defined?

(8) store and fetch conversion functions (how are they best used and implemented?)

(9) inheritance: single or multiple?

8. References

[Pf87] J. L. Pfaltz and et.al., Basic Database Concepts in ADAMS (Advanced DAta Manipulation System): Language Interface for Process Service, IPC Technical Report 87-001, UVA Dept. of C.S., November 1987.

[PfF] J. L. Pfaltz and J. C. French, Abstract of Proposed Paper: The Multi-faceted Role of Sets and Functions in ADAMS, UVA Dept. of C.S., October 1987.

[PWF88] J. L. Pfaltz, J. C. French and J. L. Whitlach, Scoping Persistent Name Spaces in ADAMS, IPC Tech. Rep. 88-003, Institute for Parallel Computation, Univ. of Virginia, June 1988.

APPENDIX A: Syntax Summary

The following is a condensed summary of the complete ADAMS language subset implemented by the AP. The syntax definition given here is exactly the same as the one in section 4.2 of this report. However, all the headings and additional text in that section has been stripped out, leaving only the BNF grammar rules (and, in the case of the host languauge dictionary interrogation routines, the function declarations).

<adams_source> ::= ('<<' <adams_stmt> '>>' | <host_stmt>)+
<adams_stmt> ::= <co_domain_def> | <class_def> | <inst_creation> | <inst_manip>
<dict_interrog> | <name_var>
<co_domain_def> ::= <co_domain_name> **is a CO_DOMAIN**
consisting of '#' <regular_expr> '#'
[<store_clause>]
[<fetch_clause>]
[<scope_clause>]
<store_clause> ::= **with store** <host_func>
<fetch_clause> ::= **with fetch** <host_func>
<class_def> ::= <user_class_def> | <set_class_def>
<user_class_def> ::= <user_class_name> **is a** <user_class_desig>
[<class_def_clause>]
[<scope_clause>]
<user_class_desig> ::= <user_class_name> | <set_class_name> | **CLASS**
<set_class_def> ::= <set_class_name> **is a SET**
of <class_desig> **elements**
[<class_def_clause>]
[<scope_clause>]
<class_def_clause> ::= (<assoc_clause> | <restrict_clause>)+
<assoc_clause> ::= **having** [<set_synonym> '='] <func_set>
<func_set> ::= <func_set_name> | <enum_func_set>
<enum_func_set> ::= '{' (<func_name>)+ '}'
<restrict_clause> ::= **in which** <quantifier> <predicate>
<quantifier> ::= '(' **for all** <variable_name> ')'
<predicate> ::= '[' <variable_name> '.' <attr_name> <rel_op> '"' <string> '"' ']'
<rel_op> ::= '==' | '!=' | '<' | '>' | '<=' | '>='
<inst_creation> ::= <attr_creation> | <map_creation> | <user_creation> | <set_creation>
<attr_creation> ::= <attr_name> **belongs to ATTRIBUTE**
with image <co_domain_name>
<value_assign_clause>
[<undef_clause>]
[<scope_clause>]
<undef_clause> ::= **with undefined** '"' <string> '"'
<value_assign_clause> ::= **value is assigned**
<map_creation> ::= <map_name> **belongs to MAP**
with image <class_desig>
<value_assign_clause>
[<scope_clause>]
<user_creation> ::= <inst_name> **belongs to** <user_class_name>
[<scope_clause>]
<set_creation> ::= <set_name> **belongs to** <set_class_name>

[**consisting of** <set_value>]
[<scope_clause>]
<set_value> ::= <set_desig> | <enum_set>
<enum_set> ::= '{' (<inst_name>)+ '}'
<inst_manip> ::= <attr_assign> | <store> | <fetch> | <map_assign> | <inst_assign> | <set_access>
<attr_assign> ::= **assign into** <attr_desig> **from** <attr_value>
<attr_value> ::= <attr_desig> | '"' <string> '"'
<store> ::= **store into** <attr_desig> **from** <host_value>
<host_value> ::= <host_var> | <host_const>
<fetch> ::= **fetch into** <host_var> **from** <attr_desig>
<map_assign> ::= **assign into** <inst_desig> **from** <inst_desig>
<inst_assign> ::= **assign into** <inst_name> **from** <inst_desig>
<set_access> ::= <insert> | <delete> | <for_all>
<insert> ::= **insert** <inst_desig> **into** <set_desig>
<delete> ::= **delete** <inst_desig> **from** <set_desig>
<for_all> ::= **for all** <loop_variable_name> **in** <set_desig> **do**
(<adams_stmt> | <host_stmt>)+
<dict_interrog> ::= <attr_view> | <map_view>
<attr_view> ::= <set_desig> **view of** <class_name>
<map_view> ::= <set_desig> **map of** <class_name>
<attr_desig> ::= <inst_desig> '.' <attr_name>
<class_desig> ::= <user_class_name> | <set_class_name> | **ATTRIBUTE | MAP**
| SET | CLASS
<inst_desig> ::= <inst_name> [<map_desig>] [<map_desig>]
<map_desig> ::= '.' <map_name>
<set_desig> ::= <set_name> | <inst_desig>
<temp_scope> ::= **LOCAL**
<permanent_scope> ::= **USER**
<scope_clause> ::= **with scope** <permanent_scope>
<name_var> ::= **name_variable** (<string>)+

Charstr class_of(inst : Charstr)
Charstr class_of_element(set_inst : Charstr)
Charstr dict_name_of(for_loop_var : Charstr)
Charstr id_of(inst : Charstr)
Charstr image_of(map_inst : Charstr)
Boolean is_class(name : Charstr)
Boolean is_dict_entry(name : Charstr)
Boolean is_element_of(inst, set_inst : Charstr)

APPENDIX B: Dictionary Organization

Although the typical AP user should not need to examine the contents of the permanent data dictionaries, this section provides the necessary implementation details if a user wishes to do so. The ADAMS dictionaries are stored in the "dict" directory (which is a sub-directory of the current working directory). The user dictionary resides in "dict/user" and the system dictionary resides in "dict/sys". The following list describes the contents of each ".dict" file in these directories. For each file, its name, the format of each entry (surrounded by square brackets), and a brief description of its contents are given. Note that an empty field (i.e., the NULL string) is stored as a zero and the symbol "$" is used as an "end of entry" marker. There are three categories of dictionary entries: co-domain definitions, class definitions, and class instances.

assoc.dict: [class name, synonym, set name]
Contains a list of all the association clauses for all class entries.

attr.dict: [attribute name, instance id, value]
Contains a list of all the attribute values for all instance entries.

class.dict: [class name, parent class, element type]
Contains a list of all class entries.

co_dom.dict: [co-domain name, regular expression, store function, fetch function]
Contains a list of all co-domain entries.

entry.dict: [entry name]
Contains a list of all entries currently in the dictionary.

init.dict
Contains the next available unique instance id. Note that this file exists only in the user dictionary (not in the system dictionary).

inst.dict: [instance name, class, instance id]
Contains a list of all instance entries.

map.dict: [map name, instance id, value]
Contains a list of all the map values for all instance entries.

restrict: [class name, attribute name, relative operator, constant]
Contains a list of all the restriction clauses for all class entries.

set.dict: [set name, value]
Contains a list of all the set values for all set instance entries.

THE FACULTY AND THEIR RESEARCH INTERESTS (1989-1990)

The faculty members of the Institute for Parallel Computation are listed below with their research interests.

REYNOLDS, PAUL F., JR. - Director, Associate Professor of Computer Science, Ph.D., University of Texas at Austin. Distributed Systems, Parallel Languages.

PFALTZ, JOHN L. - Associate Director, Professor of Computer Science, Ph.D., University of Maryland. Data Management, Graph Theory, Parallel Data Access.

PRATT, TERRENCE W. - Research Professor of Computer Science, Ph.D., University of Texas at Austin. Programming Languages and Environments, Theory of Programming, Parallel Computation.

BROWN, DONALD E. - Associate Director, Assistant Professor of Systems Engineering, Ph.D., University of Michigan. Inductive Systems, Data Analysis, Decision Support, Design Aiding.

RICHARDS, DANA S. - Assistant Professor of Computer Sceince, Ph.D., University of Illinois. Analysis of Algorithms.

MARTIN, WORTHY N. - Assistant Professor of Computer Science, Ph.D., University of Texas at Austin. Computer Vision, Robotics, Graphics.

SON, SANG H. - Assistant Professor of Computer Science, Ph.D., University of Maryland. Database Systems, Distributed Real-Time Systems, Database Prototyping.

COHOON, JAMES P. - Assistant Professor of Computer Science, Ph.D., University of Minnesota. Design and Analysis of Algorithms, Design Automation, Parallel Algorithms.

FRENCH, JAMES C. - Senior Scientist, Ph.D., University of Virginia. Data Management, Programming Environments, Performance Measurement.

STEWART, BRADLEY S. - Senior Scientist, Ph.D., University of Virginia. Terrain Modeling, Heuristic Search, Decision Analysis, Parallel Algorithms.

INSTITUTE FOR PARALLEL COMPUTATION RECENT TECHNICAL REPORTS

TR #	TITLE	AUTHORS	DATE
87-001	"Basic Database Concepts in ADAMS (Advanced Data Manipulation System): Language Interface for Process Service."	Pfaltz, John Son, Sang H. French, J. et al	November 30, 1987
88-001	"Compact O-Complete Trees: A New Method for Searching Large Files."	Orlandic, Ratko Pfaltz, John	January 26, 1988
88-002	"Reliability Mechanisms for ADAMS."	Son, Sang Pfaltz, John	March 20, 1988
88-003	"Scoping Persistent Name Spaces in ADAMS."	Pfaltz, John French, James Whitlatch, Jenona L.	June 28, 1988
88-004	"Implementing Set Operators Over Class Hierarchies."	Pfaltz, John	August 5, 1988
88-005	"Implementation of an ADAMS Prototype: The ADAMS Preprocessor (AP)."	Klumpp, Cathleen Pfaltz, John	August 9, 1988
88-006	"The 1988 Parallel Sorting Bibliography."	Richards, Dana	August 25, 1988
88-007	"A Spectrum of Options for Parallel Simulation."	Reynolds, Paul	September 9, 1988
88-008	"A Neural Network Implementation of a Correspondence Processing Algorithm."	Barker, Allen Brown, Donald Martin, Worthy	October 9, 1988
88-009	"A Prototyping Environment for Distributed Database Systems: Functional Description."	Son, Sang H. Ratner, Jeremiah Chang, Chun-Hyon	October 9, 1988
88-010	"A Global Time Reference for Hypercube Multicomputers."	French, James C.	October 10, 1988
88-011	"A Procedure for Generating Source Weights in Group Consensus Problems."	Brown, Donald E. Mostaghimi, Mehdi	December 12, 1988
89-001	"A Bibliography of Heuristic Search."	Stewart, Bradley	January 30, 1989