The ADAMS Preprocessor

Paul K. Baron

IPC-TR-89-009
December 4, 1989

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

This research was supported in part by DOE Grant #DE-F605-88ER25063 and JPL Contract #957721

Abstract:

ADAMS is a database language that has been developed by the ADAMS group of the Institute for Parallel Computation, of the University of Virginia. ADAMS deals not only with a persistent database, but also a persistent name space. The ADAMS language consists of statements that can be embedded within traditional programming languages, such as C, FORTRAN or Pascal. This allows programmers to still use a language that they are familiar with, avoiding many problems inherent in stand alone database languages.

This document describes the implementation of a preprocessor for embedding the ADAMS language within C. The preprocessor accepts a C program with embedded ADAMS statements, and translates it into a pure C program which can then be compiled by the cc compiler. The elements of the preprocessor are detailed, along with the interaction that the preprocessor has with other elements of the ADAMS system. Research issues arising from the persistent name space of ADAMS are also discussed.

1. Overview

This technical report describes the implementation and usage of the preprocessor for the ADAMS (Advanced DAta Management System) database language developed by the ADAMS group of the Institute for Parallel Computation, of the University of Virginia. The ADAMS language is described in great detail in a separate document [PFG89b]. As such, only a brief overview of the language will be presented in this report. Readers with questions on the details of the language itself are urged to refer to that report.

ADAMS is a database language with two major research goals: 1) to gain insights into developing database systems that can exploit the potential of parallel processing machines; 2) to explore the concept of a persistent name space for elements in a database system.

ADAMS consists of database manipulation statements which can be embedded within a traditional host programming language. In this version of the preprocessor the host language is C. The run time system is implemented in C++ on a VAX 8600 running 4.3 BSD UNIX†. It is intended that future preprocessors will be developed to embed ADAMS in other host languages such as Pascal, FORTRAN, and Ada‡.

This document should give the reader an understanding of how the preprocessor works, what it accomplishes, and how it was implemented. It should also serve as a good starting point for those who will modify the current preprocessor or who will actually develop the versions for other languages.

1.1. ADAMS language fundamentals

ADAMS is a database language that provides access to permanently stored, or persistent data. The data elements may have names by which they can be referenced in much the same way that variables can be named in a C program. A major feature of ADAMS is that the names for

† UNIX is a trademark of AT&T Bell Laboratories.

‡ Ada is a trademark of the Department of Defense.

these elements can also be persistent.

ADAMS code is embedded within a host language such as C, FORTRAN or Ada. This provides a common data interface for these host languages, enabling all users to write application programs in their preferred language, while permitting all to access the same data.

The guiding philosophy of ADAMS is simplicity. Many database systems become cluttered by inflexible, complex constructs, arising from patches to provide for elements left out of the basic system. FORTRAN and COBOL illustrate traditional programming languages that have grown cluttered over the years. ADAMS, on the other hand, is composed of a handful of fairly simple constructs, which can be used as building blocks to create more complex database configurations.

To obtain a background knowledge in the ADAMS language that will be needed to understand the development of the preprocessor, the following areas will be examined:

- The basic ADAMS constructs
- The concept of unique ids
- Naming of ADAMS elements
- Unnamed elements—ADAMS_vars
- Interaction between ADAMS names and the host language

The basic constructs of ADAMS are: class, codomain, attribute, map, and set. A brief explanation of each follows:

Class: This is similar to a type definition in other programming languages. It describes the properties of instances of the type. Elements, also referred to as objects, are specific instances of a class, similar to instantiations of a C structure definition, but do not take up storage as the C structures do. A C structure takes up a number of bytes equal to the sum of the size of the components and appropriate offsets for word boundaries. In ADAMS an element is more conceptual; it exists only as a pointer to its components (attributes and maps—two of the other constructs).

Codomain: This is a set of permissible data values. From the users' viewpoint they are

character strings.

Attribute: This is much like a field of a record in a traditional record and file database model. An element may have a set of attributes associated with it, each of which has a codomain value for an image.

Map: This is similar to an attribute, but the image of a map is another ADAMS element instead of a codomain value. It is used to define relationships between ADAMS elements.

Set: This is the basic ADAMS aggregate. It is a proper mathematical set; there is no ordering on the elements of the set. The elements of a set must all be from the same class.

Every element created in ADAMS has a tag associated with it known as its *unique id* (also referred to as a *uid*). As the name suggests, this tag is unique for all elements ever created. Even when an element is deleted its unique id can never be used for a new element. The unique id is the means by which ADAMS elements are retrieved. However, the ADAMS user never sees these unique ids. He may use the name of an element or a dummy variable, such as an iterator in a FOR_EACH loop. The values of the unique ids are obtained through the preprocessor or run time system.

The name of an element in ADAMS can be persistently associated with that element even after the program that created them has finished. There is a hierarchy to the name space, similar to the hierarchy of file permissions in UNIX (user, group, world). ADAMS has 3 persistent levels: *system*, *task*, *user*, and one non-persistent level: *local*.

This naming scheme differs from traditional databases where only elements of the schema—attributes and relations—are named. A relational database has no names for its individual tuples or records; the elements are accessed by a key, a part of the record itself. If the value of the key changes the record must be accessed in a different manner, through the new key. In ADAMS, if the attributes and maps of a named element change, the element can still be accessed in the same manner as before. To an ADAMS user the elements are referenced by a name, but

the actual generated C code references them by their unique ids. This is an important matter, as unnamed elements are also referenced by their unique ids. Elements can be placed into a set without ever naming the individual elements; to retrieve the elements, only their unique ids must be accessed.

While any element in ADAMS can be named, there is no requirement that an element be named. Often it is desirable for an element to be instantiated, have maps/attributes assigned to it, and be inserted into a set, without there ever being any need to ever refer to the element by name. References to the element will be made via a loop through all the elements of that set. ADAMS provides a construct known as an *ADAMS_var* for this purpose. These elements still have a unique id, so the run time system views them the same as named elements.

There is also a mechanism for linking in names with host variables. *VAR* variables in ADAMS are host variables of type "string", whose *value* will be interpreted as an ADAMS name.

1.1.1. ADAMS as an embedded language

Since ADAMS is an embedded language it must be possible to determine at all times whether an embedded or host language statement is being scanned. ADAMS solves this by delimiting all ADAMS statements by "<<" and ">>". Only characters within these delimiters need be tokenized and passed on to YACC; all others can be copied directly to the C output file.

An important issue is that of separation of the ADAMS code from the host language statements. Since only characters within the ADAMS delimiters are scanned and passed to the parser, the C statements in the host language are ignored by the preprocessor. Any information in the surrounding C code, such as host variable names, and size of C buffers used to store attributes, is lost.

1.2. Preprocessor fundamentals

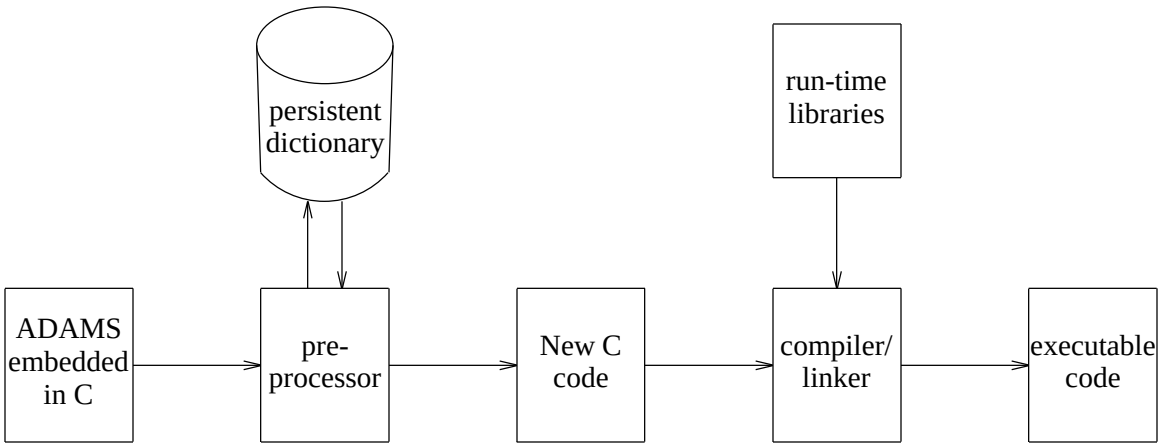
The purpose of the preprocessor is to convert a source program containing both C and ADAMS statements into a source program consisting of C statements alone. To obtain an overview of the preprocessor several issues will need to be addressed here:

- The basic elements of the ADAMS preprocessor.
- The implications for the preprocessor of ADAMS being an embedded language.
- Brief history of the ADAMS preprocessor.
- Interaction with other ADAMS modules.

1.2.1. The basic elements of the ADAMS preprocessor

The role of the preprocessor in the overall scheme of ADAMS is shown in Figure 1-1. The user embeds ADAMS statements within a C program which is input to the preprocessor. The preprocessor examines the source program and replaces ADAMS statements with C code to perform the desired function of the ADAMS code, while passing the original C code along intact. The persistent dictionary is accessed at parse time to aid in code generation.

The preprocessor creates a C program which can then be compiled and linked in with other ADAMS run-time code to produce an executable program. This is a simplification of the



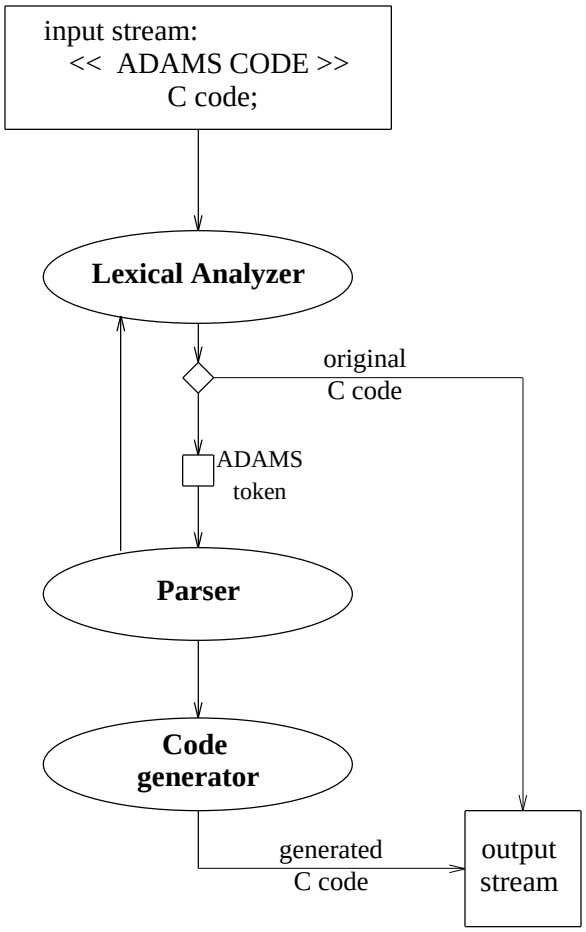
Overview of the preprocessor
Figure 1-1.

compilation/link phase, as will be seen in the section on C/C++ interface problems.

The components of the preprocessor are the lexical analyzer, the parser, and the code generator. They are illustrated in Figure 1-2.

The lexical analyzer takes the input stream of C and embedded ADAMS, and emits tokens to the parser. User written C code is sent unchanged to an output file. The UNIX tool LEX was used to write the lexical analyzer.

The parser is the centerpiece of the preprocessor. Its purpose is to determine whether or not the ADAMS portions of the input constitute legal ADAMS statements, and if so, to make



Elements of the preprocessor
Figure 1-2.

appropriate calls to the code generator, which will emit C code to perform the function of the ADAMS code. The parser was written using the UNIX tool YACC, which creates a C program that simulates a deterministic finite automaton that evaluates the stream of tokens received from the lexical analyzer and eventually accepts the input as valid, or detects an error (with attempts to recover and continue parsing).

Once the parser recognizes a valid ADAMS statement, the code generator is invoked. There actually is no one code generator entity as there is one parser and one lexical analyzer, but a set of functions, one for each ADAMS statement, that actually generate C code. This set of functions, along with a number of specialized functions that they require, constitutes the abstract entity known as the code generator.

The code generator emits C code in place of each ADAMS statement it processes into the same stream that the lexical analyzer outputs the original C code. As it processes each ADAMS statement, the code generator emits C code into and output stream. The lexical analyzer emits the original C code into this same output stream. Since the code generator does not work on ADAMS code at the same time that the lexical analyzer emits regular C code (see decision point that the lexical analyzer reaches in Figure 1-2), there are no conflicts between them in the output file. Eventually the users' source program is exhausted and preprocessing ends with the new C program as output.

1.2.2. Brief history of the ADAMS preprocessor

The current preprocessor is for the most recent version of ADAMS. A prototype was developed for an earlier version of ADAMS [Klu88]. The earlier version was purposely designed to be a prototype of a real preprocessor, with the goal being to rapidly bring a version up and running, in order to test out various constructs of the ADAMS language, and get a feel for what would and would not be useful features of the language. From that viewpoint the prototype worked out very well. There were substantive revisions made to the design of ADAMS as a

result of testing ADAMS programs with the prototype.

This preprocessor is more complex than that first prototype, and has taken more time to develop, for several reasons:

(1) The prototype was based on only a subset of the ADAMS language. Until then ADAMS was purely theoretical; there was no way to test out which elements would work. There was little incentive to write ADAMS programs since there was no way to run them, thus little critical analysis of the language itself was performed. There was a great need to get something working, and a subset of ADAMS sufficed.

(2) A rudimentary dictionary/data file system was established. The dictionary and data were stored together in the same files. This clearly would not be practical for a real system, but was no great problem since it was only a prototype. Much of our work has been concerned with issues related to the dictionary, and the persistence of ADAMS elements.

(3) There is a problem associated with the delimiters used for ADAMS when it is embedded within C. This was essentially ignored in the prototype, but had to be handled in this version. This problem will be discussed later.

The lexical analyzer that is used in the current preprocessor is based upon the one used in the prototype. However the existing code had to be greatly augmented to resolve recognition of the full set of keywords, and to handle the problem with the delimiters.

The ADAMS language itself is quite different from that used by the prototype. Numerous extensions and modifications have been made in the last year. Consequently there was little point in trying to build upon the earlier version of the parser; an entirely new one was designed.

1.2.3. Interaction with the dictionary

The dictionary is an entity whose function is to keep track of names of elements, associated unique ids, and definitions of classes. The dictionary module is the most important one that the

parser must interact with. The other modules (low-level storage manager, unique id manager) have importance to the preprocessor, but they are called indirectly, as with the dictionary module utilizing the unique id manager.

The interaction between the parser and the dictionary occurs when the preprocessor needs to obtain from, or give to, the dictionary information about an element. Information that the preprocessor may wish to obtain or give includes the unique id of a named element, the superclass(es) of an element, whether an element exists already or not at parse time, and whether or not an element is a subscripted variable, and if so, the number of dimensions it should have.

A more detailed examination of the dictionary levels is needed now. There are 3 persistent levels to the dictionary: *user*, *task*, and *system*, and one non-persistent level, *local*. Much as with UNIX permissions, *user* level allows a user to define his own classes and create names for his own objects without affecting other users. *Task* level is above this. A *task* is a collection of users engaged in some common work. Names at the *task* level can be seen by all users in this *task*, much as with the group level in UNIX. A user can be a member of any number of *tasks*. Above this is the *system* level, whose names are available to everyone. When names are placed into these 3 levels they stay there until they are specifically deleted or rescoped. The termination of the process in which they are created or manipulated does not remove either the names or the entities they denote.

The *local* level is similar to the user level in that it is accessible only to one user. However it does not have a persistent scope: when the program that created entities in this level ends, both the names and associated entities are deleted. The purpose of having this level is to serve as a "scratch pad" for creating temporary elements and classes on the fly, without having to worry about the names conflicting with elements that already exist in the persistent name space.

1.2.4. Interaction with the index manager

The index manager keeps track of information for unique ids such as which are members of which ADAMS sets, what are the reference counts for each unique id. The index manager provides a buffer between the views of elements as named entities and their representation in storage. The index manager sees only unique ids, never ADAMS names, and makes calls to the low-level storage manager.

An important function of the index manager is to keep a reference count on all named elements. A reference count in ADAMS is similar to the link count for UNIX files, where a file's link count is incremented every time a user creates a new link to it. This gives the user the perception of having his own copy of the file, without using up space to copy the file; only one copy actually exists, so all users with links to it are affected by changes to it. When somebody deletes his copy of a linked file it doesn't delete the actual file, but only his link to it; the file itself is deleted only when the last user to have a link to it deletes it. ADAMS reference counts work similarly, though they deal with ADAMS elements, not files. Every time a program is executed and creates a persistent reference to an element (for instance puts element x into a persistent set) the reference count for that element is incremented. As programs "delete" the element (for instance some user removes x from his set and tries to delete it) all that occurs is that the reference count is decremented. It is not until the count is one and a program attempts to delete it that the element is actually removed.

For the most part reference counts are used by the run time system to maintain database consistency. In a later section we will examine parse time/compile time conflicts that may be affected by the reference counts.

1.2.5. Interaction with the low-level storage manager

The responsibility of the low-level storage manager is to handle the details of actual storage of ADAMS elements onto permanent storage, providing an interface that will allow for various

patterns of data migration without disturbing higher levels of the ADAMS system. In short, the details of how data is stored are abstracted away from other areas of ADAMS.

The preprocessor has no direct communications with this manager—the preprocessor merely issues calls to the index manager to store or retrieve information through the unique ids. The details of how such storage/retrieval are implemented are of no concern to it.

There is one area in which there could be a connection, though it has not been implemented yet—that of clustered attribute sets, in which the user could give a hint that certain pairings of attributes or maps are generally going to be accessed in an all or none fashion, thus justifying the storage of these elements together on the same device, much like a tuple is in a relational database.

For more details about the ADAMS low-level storage manager, refer to [Jan89].

1.3. Sample ADAMS program

In the next sections, we will examine the operation of the preprocessor in greater detail by thoroughly examing the lexical analyzer, parser, and code generator. The focus will be on understanding these components and their interactions with each other and with other components of ADAMS. We will also see some problems that were encountered in development of the preprocessor and changes that were needed to the ADAMS language due to lexical/parsing considerations.

To help illustrate the fundamentals of the preprocessor, the following sample ADAMS/C code will serve as a running example in the body of the report. A more extensive example, along

with its translation into C, will be provided in appendix 5.

```
/*  sample ADAMS/C code  */

#include <stdio.h>

main ()
{
int   i;
char  new_elem[30];
char  name[30];

<< open_adams 3  >>
<< adams_var x >>

for (i = 0; i < 10; i++)
      {
      scanf ("%s", new_elem);
      << insert var new_elem into old_set >>
      }
<< insert abc into old_set >>
<< delete ADAMS_element[3, 5] from old_set >>

<< FOR_EACH x in old_set do
      << fetch from x.attr_1 into name >>
      printf ("name of next element = %s\n", name);
>>

<< close_adams 3 >>
}
```

The operations in the program are reasonably simple. The open_adams statement opens some files and performs some initialization. The C for loop scans for character strings which the insert statement takes as the names of ADAMS elements, which it then tries to insert into a (presumably) previously existing set. After the C loop, a previously defined element *abc* is inserted into the set, and a subscripted element, ADAMS_element [3, 5] is removed from the set. Next all the elements of this set are retrieved, and one attribute (*attr_1*) is printed out for each element. The close_adams statement then closes some files. The rationale behind this program will become clearer when we examine the C code that is generated for this program.

2. The Lexical Analyzer

First we will look at the lexical analysis phase of the preprocessor, the first operation performed upon the users' input text by the preprocessor.

2.1. Fundamentals of LEX

It is important to understand the role of the lexical analyzer in the preprocessor, and the relationship between LEX and YACC programs. Without this understanding, it will be difficult to understand the problems that were involved in developing the preprocessor.

The purpose of a lexical analyzer is to take an input text stream, scan it and break it up into more manageable pieces known as tokens. For example, for the following line from the sample program:

<< insert abc into old_set >>

The lexical analyzer will send the following token stream:

STMT_BEGIN, INSERT, ACTUAL_NAME, INTO, ACTUAL_NAME, STMT_END .

The LEX program specifies a set of rules governing how the input stream gets broken down into tokens. In the example above, in all cases except those corresponding to the ACTUAL_NAME tokens ("abc" and "old_set"), there were LEX rules hard-coded for exact matches. One rule specifies that the character sequence "<<" generates the STMT_BEGIN token, the literal word "insert" (actually case-insensitive for each letter) generates the token "INSERT". There is a more general rule for deciding what gets tokenized as ACTUAL_NAME. Any character sequence that does not match a previous rule, which starts with a letter and consists solely of letters and digits, will generate the ACTUAL_NAME token.

The parser (YACC program) calls for one token at a time and upon receiving each, makes a decision regarding the validity of the input program. By default, any characters sent to the lexical analyzer that do not get tokenized simply are sent to the C output file.

2.2. ADAMS influence on the lexical analyzer

Being an embedded language, ADAMS creates some difficulties for LEX, necessitating the use of some of the more advanced features of LEX. The lexical analyzer has to be sensitive as to whether it is scanning ADAMS or C code. The same character string may have different meanings to LEX in the two environments. For example, the string "MAP" is a keyword if seen in ADAMS, but if outside of ADAMS, it is merely a C construct, and is not even tokenized by the lexical analyzer.

To handle the problem of having to know which environment LEX is in, it is necessary to use LEX *start conditions* on rules. These essentially simulate a C switch statement. This concept consists of LEX knowing a set of available states which it can be in at any time. The particular state it is in is dependent upon its input. The lexical analyzer starts off in a default state. Once the STMT_BEGIN is recognized, an <ADAMS> state is entered. All of the rules for ADAMS keywords are dependent upon being in this ADAMS state, so as not to get tokenized erroneously if they also happen to occur the in C code.

There are two exceptions to this. First, if legal C code fools the lexical analyzer into thinking that it's handling ADAMS code, then ADAMS keywords appearing in the C code (such as "map" in the example) will get tokenized. However, eventually the error handling productions will be reached, and the original characters will be sent to the output stream.

The other exception concerns the STMT_END token. Normally, the lexical analyzer would be in its ADAMS state, processing an embedded statement when it encounters the STMT_END token. But this need not always be true. The lexical analyzer must correctly handle STMT_END in either state. This is necessary to allow ADAMS statements to be intermingled with C statements within the body of an ADAMS looping statement. The FOR_EACH statement allows any mix of host and ADAMS statements in its body. Not requiring the lexical analyzer to be in the <ADAMS> state to recognize STMT_END permits it to exit the <ADAMS> state after recogniz-

ing the STMT_END. The following example will illustrate this.

```
<< FOR_EACH x IN y DO
        << FETCH x.map1 INTO z >>
        printf ("%s \n",z);
>>
```

The first line is recognized as a header of the FOR_EACH statement. ADAMS and C statements follow this, and then a closing ">>" for the FOR_EACH statement. The lexical analyzer must turn off the <ADAMS> state after seeing "do", so that the following C statements are not tokenized. The fetch statement turns the <ADAMS> state on again, then back off at its end. After the next line of C code the <ADAMS> state is still off. So it is necessary to be able to recognize the STMT_END for the FOR_EACH statement while LEX is **not** in the <ADAMS> state.

There is one other function that the lexical analyzer must handle for the ADAMS language. The original characters (lexemes) that were used to form a token are still available after being tokenized if desired. LEX has facilities to handle passing this information to YACC, but this information is normally lost to LEX after it starts work on a new token. In the preprocessor's lexical analyzer, this information must be retained for the duration of an ADAMS statement for possible error handling by the parser.

3. The Parser

The parser is the single most important component of the preprocessor, so understanding its operation is essential. Since it is a YACC program, a short explanation of YACC is in order. For more a detailed explanation see [Joh78, KeP84] or any UNIX documentation.

A YACC program takes an input file, parses it, and determines whether or not the input followed a set of rules specifying the legal input. YACC decides whether or not the input is a string recognized by a specific grammar.

The main part of a YACC program consists of a set of productions in BNF. As expected in BNF, there is one start symbol, from which all allowable strings are eventually derived. Each of the right hand sides of the productions consist of tokens and/or other production names such as:

```
insert_stmt:    INSERT element_desig INTO set_desig
```

where the names in capital letters are terminal symbols and the rest are production names. From the viewpoint of the YACC user, the purpose of the program is to accept those strings (i.e., those input programs) that follow the grammar, while rejecting any other input.

The YACC specifications are transformed into a C program which creates a pushdown automaton that actually implements the parsing, but the details of that process are not necessary for understanding the ADAMS preprocessor.

The goal for a YACC program is to take the stream of tokens it receives, recognize various sequences of them as forming the right hand sides of productions, reduce those to production states, then reduce the combination of production states and remaining tokens, eventually resulting in a specific production, the start symbol, being reduced.

This process is best illustrated by examining how a specific ADAMS statement is parsed. To do so, we will have to examine the levels of productions that are higher than a statement level. The top-level production in the parser is:

```
adams_code:                          /* empty string */
                        |   adams_code adams_stmt
                        ;
```

This means that the *adams_code* production is satisfied by either the *empty* string or itself followed by an *adams_stmt*. Since there is left recursion here there can be any number of *adams_stmt* strung together in forming *adams_code*. At the start, before YACC even gets a token, the *empty* string reduces to the production *adams_code*. If the next ADAMS code that follows reduces to an *adams_stmt* (which it should, else it is an error; *adams_stmt* covers all legal ADAMS statements) then the combination of existing *adams_code* and the *adams_stmt* reduces to *adams_code* again. This process iterates as long as there is source code to parse.

The next level is the *adams_stmt* production, which has 35 choices on the right hand side, corresponding to each of the possible ADAMS statements:

```
adams_stmt:     STMT_BEGIN  insert_stmt  STMT_END
                |  STMT_BEGIN  delete_stmt   STMT_END
                |       /*  and 33 other possible statements—listed in appendix 2 */
                |       /*  and 3 possible error states  */
                ;
```

We will now show how to parse the following line from the example program:

<< INSERT abc INTO new_set >>

The option of interest from the *adams_stmt* is the first line, using the *insert_stmt*. These additional productions will be needed to explain how to parse it.

```
insert_stmt:        INSERT  element_desig INTO  set_desig
set_desig:          element_desig
element_desig:      element_name
element_name:       actual_name
actual_name:        CHAR_SEG  |   actual_name UNDER_CHARS
```

First we assume that the parser has just recognized an *adams_code* production. We have already seen that the token stream for this will be:

STMT_BEGIN, INSERT, CHAR_SEG, INTO, CHAR_SEG, UNDER_CHARS, STMT_END.

Upon seeing the STMT_BEGIN, the first element of the *adams_stmt* is satisfied, though there are 35 possible right hand sides that could be satisfied with the input to be seen. The next token,

INSERT, narrows it down to one choice, that with the *insert_stmt*. The CHAR_SEG which comes next is reduced to an *actual_name*, then to an *element_name*, and that in turn to an *element_desig*. The INTO token satisfies the next desired element. The second CHAR_SEG and the UNDER_CHARS also reduce to *actual_name* then to an *element_desig* also, but the *element_desig* then reduces to *set_desig*. These steps fulfill the requirements for reducing the *insert_stmt*.

The first two parts of the right hand side of *adams_stmt* are matched, and the final token STMT_END matches the last element. At this point the token sequence shown has been reduced to *adams_stmt*, which along with the existing *adams_code*, recursively become another instance of *adams_code*.

The other important concept in YACC is that of semantic actions. This goes beyond simple recognition of valid strings, and allows actions to be taken as a result of the input. These actions are specified through C code, and are placed on the right hand side of the productions, between tokens/production names, or more commonly at the end of a production. It is through these semantic actions that the "real" work of the preprocessor is done. Without semantic actions the only accomplishment would be the discovery of whether or not the input string was a legal program. The semantic actions call functions that translate the ADAMS statements into C code based upon what the parser has seen to date.

The semantic actions make calls to the persistent dictionary in order to determine whether or not elements exist, to find out the types of elements, and to obtain the unique ids of elements. Using this information the proper C calls can be generated. To understand these C calls that the preprocessor generates, we need a more detailed description of the dictionary.

3.1. Language recognition problems

Once the syntax became reasonably stable, developing the preprocessor was quite straight forward. However there were a few small problems whose description may be instructive.

The characters delimiting ADAMS statements are "<<" and ">>".

```
i = 3;
<<        lock x   >>
y = foo (i);
```

In most cases this allows for the easy recognition of ADAMS statements. Once LEX sees the characters "<<" it returns a token STMT_BEGIN to the parser, signifying the start of an ADAMS statement. YACC then expects the next tokens to be the body of an ADAMS statement, followed by the token STMT_END, corresponding to input characters ">>".

However problems arise with valid C code such as:

```
i = 3;
x = y << z ;
y = foo (i);
```

The code fragment above is syntactically correct, but a simple-minded LEX/YACC combination would find an error with it, as the "<<" would be interpreted as the start of ADAMS code instead of the C operator.

A natural question is "Why use such delimiters if they cause so much trouble? Why not choose something else?". This is a valid point: picking different delimiters, say "!^" and "^!" as the start and end markers, would eliminate these problems.

Or would it? ADAMS is intended to be embeddable in any programming language. No matter what delimiters are picked, if they are at all aesthetically bearable, there is likely to be some programming language out there in which they are significant. A hideous combination of characters such as "<$!#" would most likely not be lexically significant in any programming language, but that is because everyone else had the sense to reject using them.

Another argument for using these delimiters is that they give just what was desired: the sense of the start and finish of a statement, in the manner in which they physically enclose code. It is difficult to find other combinations of characters that are even reasonably aesthetically pleasing, much less so functional as well. We decided to find a way to use these delimiters within the

C language.

A solution was found through extra interaction of the YACC and LEX programs. Every time that LEX finds the token "<<" (STMT_BEGIN) it starts keeping track of the actual character strings that were tokenized; normally this information is lost to LEX once it emits a new token. As tokens are sent to YACC, eventually either the parser recognizes a legal ADAMS statement, or enters an error condition.

Originally there was only one error handling statement:

```
adams_stmt:    error    STMT_END
                        {  /*  semantic actions */  }
```

("error" is a special name to YACC, meaning simply that the tokens sent to it led to an unrecognizable string. Instead of simply aborting, an attempt to recover from the error can be made, just as a normal compiler should continue after finding the first error in a program). Once YACC hits an error state, it is permitted to consume tokens until it finds STMT_END (">>"), at which point it prints an error message, exits the error state, and starts processing the rest of the string.

To handle the conflicts between the ADAMS and C usage of the delimiters, several more error statements were introduced:

```
adams_stmt:    error ';'
adams_stmt:    error ')'
```

Every legal C statement that uses a left-shift must eventually contain a ';', ')', or '>>', and each of these cases would be handled by one of the error statements. Instead of printing an error message, those "errors" ending in ';' or ')' force the output of the list of strings that LEX had been saving since it saw the STMT_BEGIN, and drop out of parsing an ADAMS statement, allowing the rest of the line of C code to be passed to the output file untouched.

For example, in this line:

```
y  =  x  <<  3  +  (z  *  MAP) ;
```

the "y = x" was passed to the output since (we assume) the parser was scanning C code, not an ADAMS statement. LEX passes the "<<" (STMT_BEGIN token) to YACC, which attempts to parse an ADAMS statement. Very quickly the parser will realize that "3 + (" is not part of the body of any legal ADAMS statement. An error state is reached in which it remains until it see the token ')'. It then emits the strings that LEX had been saving since the STMT_BEGIN token was seen. After this the parser stops looking for parts of an ADAMS statement, outputting the C code as is, until another "<<" (STMT_BEGIN) is seen again. Until a new starting delimiter is seen, LEX passes the code to the output file untouched. In the example above, "y = x " is passed on as part of C, "<< 3 + (z * MAP)" is passed as part of the "error" state, and the ";" is passed on as part of C again. So the whole legal C code is passed on as seen. The YACC program returns a value to its calling program indicating acceptance or rejection of the input string. In this case no action taken in these cases to indicate that there was an error in the program, since there actually was none.

This dependence between the LEX and YACC programs for handling C code is undesirable, as it requires changes to be made to the lexical analyzer for tokenizing ADAMS embedded in other languages. However the changes needed are minimal, and not too high a price to pay for using the desired delimiters. LEX and YACC purists would object to this error handling on the basis that the lexical analyzer and parser should not have any intimate knowledge of the other, that each has a function that is not dependent on the inner workings of the other. In practice this is often violated.

One additional advantage of having set up the handling in this manner is that it makes it very easy to set up an option to print out the original ADAMS statements inside of comments right after the corresponding C code emitted for the statement in the C file. An option on the command line for the preprocessor allows these comments to be added for debugging purposes. All that was necessary to add this feature to the preprocessor was to have a conditional statement at the highest level of the parser (recognizing a whole ADAMS statement) that would, if desired,

print out the list of strings stored by LEX at the end of each legal ADAMS statement (enclosing them within comment symbols), as opposed to emitting them only when in an error condition.

3.2. C/C++ interface problems

The run time ADAMS system has been implemented in C++, an object-oriented programming language that is a superset of C [Str87] The preprocessor assumes that C is the host language and must accept and generate only legal C code. Although C and C++ are similar, they are not the same language. As will be seen only pure C code can be generated, and it must interface with the C++ run time system. The same situation will occur when we write preprocessors for ADAMS statements embedded in Pascal, FORTRAN and other host languages.

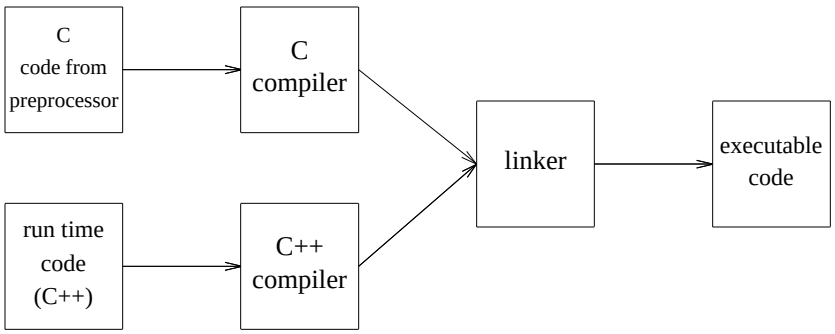
The decision was made to implement the ADAMS system in C++ mostly due to its reputation for facilitating the rapid development of large software systems. The original approach was to emit C++ code from the parser and intermingle it with the users' C code. This appeared to be sensible, as C++ is generally thought of as a superset of C, with the only compatibility problem with C being that C++ uses new keywords. However we found numerous instances where C++ would not compile legal C programs.

One of the problems was that C++ expects a different style for function argument declarations than C, and the two are incompatible. The +V option for CC (the C++ compiler) on 4.3 BSD UNIX gets around that problem, but the man page for CC states that that option is not guaranteed to be supported in the future, making dependence upon it risky. Another problem is that the manner in which argv is declared in the program determines whether or not argv can be incremented. The declaration "char **argv" is acceptable, while "char *argv[]" is not. Users who write legitimate C programs with the second declaration, and then attempt to increment argv would get compile time errors from CC, such as "attempt to alter the value of a constant expression".

This led to the realization that if the preprocessor emitted C++ code, then the user would have to use C++ as the host language for this system, else there would be no guarantee that his host code would survive intact. It was felt that although C++ is becoming more popular, it still has a small support group, in comparison to C. Therefore it would be better to use C as the host language, and to emit only C code in place of the ADAMS statements. By using C as the host language and C++ for the run time implementation, we are already working through many of the problems that will occur when embedding ADAMS in FORTRAN.

This strategy created problems elsewhere though. Since the preprocessor generates a file with only pure C code, the output file cannot access any structures or objects defined in C++ code in other modules. Any direct references to C++ objects would force the code to be compiled by CC.

Figure 3-1 illustrates the changes to the compile/link phases due to the addition of modules written in C++. The two boxes in the leftmost column represent the C code from the preprocessor and the C++ code for the run-time system. The preprocessor C file is run through the normal C compiler, resulting in a .o file, while the set of support routines are run through the C++ compiler, also resulting in a .o file. All of these .o files are then linked together, resulting in the final executable code.



Effect of C++ addition on compile/link phases
Figure 3-1.

Unique ids are represented in the C++ programs by a C++ class which gets translated into a C structure. Since there is only C code in the preprocessor output, obviously the C++ class cannot be used there. One option was to use the C translation of the unique ids in the C program. There were problems with this, as there would have to be C calls to the C++ constructors and destructors and to the C++ member functions, and consistency would have to be maintained between the C++ and C definitions in the various programs. To reduce the burden on the preprocessor, it was decided that unique ids will be seen as strings. As strings they can be passed back and forth between functions that require them. There are C++ routines that convert between the string representation and the C++ class version of unique ids.

An alternative strategy that was considered was to decide what C++ calls the ADAMS statements should map onto, but instead of actually using them, translate all the C++ calls directly into their equivalent C statements. This differs from the strategy of using the translated version of unique ids in that all the C++ code would be translated. This would amount to doing the work of the C++ front end, and was seen as being overly ambitious for the first version of ADAMS.

4. The Code Generator

In order to make the task of code generation for ADAMS statements manageable, and the code generator maintainable, it was necessary to create a number of levels of abstraction in the code generator.

The first occurs in the parser, where in almost every case (see the section on code generation for loops as the exception) code generation is abstracted into a single call to the code generator which occurs at the statement recognition level—that is, where the choices on the right hand side of the production in the yacc program are the several dozen ADAMS statements. This serves to fairly cleanly separate the functions of parsing and code generation.

The next level of abstraction occurs within the statement level, where one must find the *uids* of named ADAMS elements. There are several checks that must be made on a named entity en route to obtaining its *uid* from the dictionary: does the element have a scope attached to it ?; is it a subscripted name ?; does it have a element designator (map/attribute) ?; is it an ADAMS_var ?; and is it a var variable? The code that obtains the uids for an ADAMS element will be considered to be an abstract entity known as the element handler.

The general procedure in generating code for a single statement is that the checks for these possibilities are made in other routines. When dealing with an element, the statement level function simply makes a call to an element handler, which will generate code to obtain the *uid*. The statement level generator is concerned only with what to do with the *uid* it gets, not the details involved in obtaining it.

In order to understand the translation from ADAMS code to C, we will look at three areas of code generation: the initial code which must be generated for all ADAMS programs; that code emitted by the statement level functions, once they have obtained the *uids* for their arguments; that emitted by the element handler in finding the *uids* of ADAMS elements.

4.1. Initial code generation for all ADAMS programs

There is an amount of C code which is generated for all ADAMS programs. This includes declarations for variables which other generated C statements will refer to, #include statements for header files from the ADAMS managers (index, dictionary, low-level storage), and #defines for constants. Currently these statements are generated in the C file created for each ADAMS program; they should probably be replaced with a single #include for a file which would already contain these statements.

All of the variables declared in statements emitted by the code generator start with the sequence "_A_", so as to reduce conflicts with user-declared C variables. Most of them are straightforward declarations needing no explanation. One that does need some explaining is *_A_args*. *_A_args* is declared as:

```
char _A_args [_A_MAX_ARGS_PER_STMT] [_A_IDLEN + 1];
```

_A_args is used to store the *uids* of the arguments in a single ADAMS statement. The defined constant _A_MAX_ARGS_PER_STMT sets a limit on the number of arguments in all statements, most notably SET_INTERSECTION and SET_UNION, which may have a variable number of arguments. The constant "_A_IDLEN + 1" is the size of a *uid* plus its null terminator.

4.2. Code generation for specific ADAMS statements

The rationale behind the code generation for all of the ADAMS statements will be explained in this section. First we will discuss some issues of coding style, then examine the transformations from specific ADAMS statements to C code.

Coding style

Some effort was put into trying to generate code in a reasonably readable manner, to aid in debugging. Effort was made to keep track of the indentions necessary for nested FOR_EACH loops for example. This effort was abandoned fairly quickly, as it was realized that the lack of

knowledge of the surrounding user C code crippled efforts to create generated C code that was easily read..

Still some effort is still made to enhance readability. If-else constructs are generated in a Pascal style indentation. All C code is generated as though the ADAMS code had started in column 1. Any ADAMS code that had been indented, say being subject to a C conditional, would not get that expected additional indentation. If the user wants nicely lined up code, he would use a utility such as cb or indent in UNIX.

One feature that has been put in for readability/debugging is that of an option to insert the original ADAMS statements into the generated C code, within comments. When this is used, the comments appear *after* the generated C code. See appendix 1 for details on usage of this feature. There are also comments that the preprocessor generates related to the C code that it generates, such as listing the argument number being processed, mentioning if is a temporary unique id for a map, and the parent class(es) of elements being instantiated. Intense debugging efforts of the generated C code will give an indication of the utility of such comments, and where other such generated comments could be of use.

Generated code

Since much of the code generated for some statements is similar to that of other statements, they will be explained together under the following breakdowns:

Open/close

OPEN
CLOSE

Declarations

ATTRIBUTE DECLARATION
CLASS DECLARATION
CODOMAIN DECLARATION
MAP DECLARATION
SET DECLARATION
SUBSCRIPT_POOL DECLARATION

Instantiations

ATTRIBUTE INSTANTIATION
ELEMENT INSTANTIATION
MAP INSTANTIATION
SET INSTANTIATION

Codomains/Subscript_pools

ADD_CODOMAIN
EXTEND_SUBSCRIPT_POOL

ADAMS_vars

VAR_DECL
VAR_ASSIGN

Set Manipulation

COMPLEMENT
INSERT
INTERSECTION
MAKE_EMPTY
REMOVE
SET_ASSIGN
SET_COPY
UNION

Looping

FOR_EACH
EXIT LOOP

Assignments

FETCH
STORE
ATTRIBUTE_ASSIGN
MAP_ASSIGN

Dictionary Manipulation

DELETE
ERASE
RESCOPE

4.2.1. Open/Close Statements

Currently the *open_adams* and *close_adams* statements do not do as much as was originally intended. They each have an associated *job_id*, but that is merely a dummy token, necessary for statement recognition by the parser, but which is thereafter ignored, since the code generator does

nothing with it.

Open:

Several lines of initializations are generated for the OPEN_ADAMS statement. These include initializations of _ADAMS_STATUS and several instances of entry_defs, and calls to the C++ main ( ) routine (in order to force C++ constructors to be called) and the *_A_attach_dict* ( ) and *_A_sym_tbl_open* ( ) routines.

The generated code is as follows:

```
_ADAMS_STATUS = 1;
_main();
_A_attach_dict (tid, uid, "open",
            "/at0/pkb4h/yacc/adams/run_tests/adamsdict", 0);
_A_sym_tbl_open ("open_sym" );
/*_A_c_check_symbol_table ();*/
_A_def.type = _A_NDEF;
_A_temp_def.type = _A_NDEF;
_A_set_def.type = _A_NDEF;
```

Note that in the generated code we have a call to *_A_c_check_symbol_table* which has be "commented out". In this initialzation phase, one should perform a variety of run time checks after openning the dictionary. This version does not. But the stub is included as a reminder for future versions.

Close:

For the CLOSE_ADAMS statement, there are only 4 statements generated: initialization of _ADAMS_STATUS, making a call to clear the entry definition, and calls to close the index manager and release the dictionary: The generated code is as follows:

```
_ADAMS_STATUS = 1;
set_deftype (&_A_def, _A_NDEF);
_A_close_indexes ();
_A_release_dict ();
```

4.2.2. Declaration Statements

Most of the code generated for declaration statements consists of calls to dictionary routines that set up an *entry definition* for the class.

An *entry definition* must be instantiated before any declaration statements are encountered, and must be initialized to be of no type. The preprocessor declared the variable *_A_def* along with it's other declarations. The **OPEN** statement does this initialization.

For each type of definition, the following pattern is followed for code generation:

—Initialize the *entry definition* to the proper type (INSTDEF, CLASSDEF, CODEF, SUBSDEF).

—Fill in appropriate pieces of the definition.

—Add the entry definition to the dictionary.

There are a number of items that can be added to an *entry definition*:

superclass
image
attribute_set
map_set
validating method
element_class
instant_class

There are also other items that could be added for the codomains, but have not been implemented yet. Refer to [PFG89a] for more detailed information on the dictionary.

First we will briefly describe each of these items, and then examine each type of declaration to see which of these may be used for each. It will be useful to have several examples of declarations on hand as the components are described:

<< **ADAMS_class ISA CLASS1 and CLASS2, having attr = {attr1_inst}, scope is system** >>
<< **ADAMS_map ISA MAP with image elem1, scope is user** >>
<< **ADAMS_set ISA SET of ADAMS_class ELEMENTS, scope is task** >>

Entry definition items:

—**superclass**

A *superclass* is a class from which an ADAMS class is derived. Superclasses may be the ADAMS base classes CLASS, MAP, ATTRIBUTE and SET, or user-defined classes. In the above example, ADAMS_class has *superclasses* CLASS1 and CLASS2. This means that anything that is an ADAMS_class has properties of both CLASS1 and CLASS2 elements.

There is one call to *_A_add_superclass ( )* generated for each of the superclasses of a class.

—**image**

An image is a class that is the "target" of a map or attribute, that is, the ADAMS entity that the map/attribute gives. A map or attribute must have one image, which leads to the code generator emitting one call to *_A_add_image ()*.

—**attribute_set** and **map_set**

These are sets of attributes or maps that are associated with a class. They are declared through a **association_clause**, which has the syntax:

association_clause := HAVING [set_name =] { list_of_attr/map_designators }

The preprocessor must interrogate the dictionary to ensure that the entities in the set in a given association clause are either all attributes or all maps. This is because this set becomes a (possibly named) ADAMS set, which may contain elements from only one class.

There can be any number of association clauses for a single ADAMS class. The preprocessor generates one call to *_A_add_attrset ( )* or *_A_add_mapset ( )* for each association clause.

—**validating method**

The ADAMS syntax ensures that a *codomain* has a clause that gives a validating method defines the allowable strings in the codomain. A generated call to *_A_add_validm ( )* enters the information for the validating method into the *entry definition*.

—**element class**

This is used for SET declarations, to name the class of elements that are allowed as members in the set. One call to *_A_add_elementclass ( )* is generated for each SET declaration.

—**instance class**

This is used for instantiations, not declarations. It signifies the class(es) of which it is an instance.

Attribute Declaration:

This must have an *image* defined, and may have any number of association clauses. The example

<< **ADAMS_attr isa ATTRIBUTE with image ADAMS_attr_image, having x = {attr1}** >>

generates the following code:

```
_ADAMS_STATUS = 1;
_A_set_deftype (&_A_def, _A_CLASSDEF);
_A_add_superclass (&_A_def, "ATTRIBUTE", _A_USER);
_A_add_image (&_A_def, "ADAMS_attr_image", _A_USER);
strcpy (_A_uid_name, ",#");
                              /* run-time test */
if (!_A_add_entry ("ADAMS_attr", _A_uid_name, &_A_def, _A_CLASS, _A_USER))
      {
      printf ("error with adding attribute ADAMS_attr to dict\n");
      _ADAMS_STATUS = 0;
      }
```

Class Declaration:

This can have any number of *superclasses* from which it is derived, and any number of association clauses. The example

<< **ADAMS_class ISA CLASS1 and CLASS2, having attr = {attr1_inst}, scope is system** >>

generates the following code:

```
_ADAMS_STATUS = 1;
                         /* declare class ADAMS_class of type CLASS1 */
_A_set_deftype (&_A_def, _A_CLASSDEF);
_A_add_superclass (&_A_def, "CLASS1", _A_SYSTEM);
_A_add_superclass (&_A_def, "CLASS2", _A_SYSTEM);
```

```
_A_c_const_uid (_A_class_uid, _A_ATTRSET);
strcpy (_A_set_uid, "-#");    /* copy uid obtained at parse time */
_A_uid_instant (_A_set_uid, _A_class_uid);
                              /* insert MAP elements into set     */
_A_d_level = _A_USER;
                              /* run-time test */
if (!_A_c_check ("attr1_inst", _A_uid_name, _A_INST, &_A_d_level))
      {
      printf ("c_check failed for attr1_inst in class_decl:\n");
      _ADAMS_STATUS = 0;
      }
_A_set_insert ( _A_uid_name, _A_set_uid);
                              /* add MAP set to definition */
_A_add_mapset (&_A_def, NULL, _A_set_uid, _A_USER, "attr");
strcpy (_A_uid_name, ".#");
                              /* add class definition to dict now: */
                              /* run-time test */
if (!_A_add_entry ("ADAMS_class", _A_uid_name, &_A_def, _A_CLASS, _A_SYSTEM))
      {
      printf ("_A_add_entry failed for class ADAMS_class of type\n");
      _ADAMS_STATUS = 0;
      }
```

Codomain Declaration:

Codomains still need some work in finalizing their semantics. The simplified version implemented allows only a regular expression to be specified that defines the allowable strings. The example << **ADAMS_codomain ISA CODOMAIN CONSISTING OF #[a-zA-Z]+#, scope is SYSTEM** >> generates the following code:

```
_ADAMS_STATUS = 1;
_A_set_deftype (&_A_def, _A_CODEF);
_A_add_validm (&_A_def, "#[a-zA-Z]+#");
strcpy (_A_uid_name, "/#");
                                    /* run-time test */
if (!_A_add_entry ("ADAMS_codomain", _A_uid_name, &_A_def, _A_CO, _A_SYSTEM))
      {
      printf ("error with adding codomain  to dict\n");
      _ADAMS_STATUS = 0;
      }
```

Map Declaration:

This must have an *image* defined, and may have any number of association clauses. The example << **ADAMS_attr isa MAP with image ADAMS_map_image, having y = {map1}** >> generates the following code:

```
_ADAMS_STATUS = 1;
```

```
            /* declare map ADAMS_attr of type ADAMS_map_image */
_A_set_deftype (&_A_def, _A_CLASSDEF);
_A_add_superclass (&_A_def, "MAP", _A_USER);
_A_c_const_uid (_A_class_uid, _A_MAPSET);
strcpy (_A_set_uid, "0#");    /* copy uid obtained at parse time */
_A_uid_instant (_A_set_uid, _A_class_uid);
                        /* insert MAP elements into set */
_A_d_level = _A_USER;
                        /* run-time test */
if (!_A_c_check ("map1", _A_uid_name, _A_INST, &_A_d_level))
      {
      printf ("c_check failed for map1 in class_decl:\n");
      _ADAMS_STATUS = 0;
      }
_A_set_insert ( _A_uid_name, _A_set_uid);
                        /* add MAP set to definition */
_A_add_mapset (&_A_def, NULL, _A_set_uid, _A_USER, "y");
strcpy (_A_uid_name, "1#");
_A_add_image (&_A_def, "ADAMS_map_image", _A_USER);
                        /* add class definition to dict now: */
                        /* run-time test */
if (!_A_add_entry ("ADAMS_attr", _A_uid_name, &_A_def, _A_CLASS, _A_USER))
      {
      printf ("_A_add_entry failed for map ADAMS_attr of type ADAMS_map_image \n");
      _ADAMS_STATUS = 0;
      }
```

Set Declaration:

This statement expects one *class_name* of elements allowed in instances of the set type, and any number of association clauses. The example

<< **ADAMS_set ISA SET of ADAMS_class ELEMENTS, scope is task** >>

generates the following code:

```
_ADAMS_STATUS = 1;
                  /* declare set ADAMS_set of ADAMS_class elements */
strcpy (_A_uid_name, "2#");
_A_set_deftype (&_A_def, _A_CLASSDEF);
_A_add_superclass (&_A_def, "SET", _A_TASK);
_A_add_elementclass (&_A_def, "ADAMS_class", _A_TASK);
                        /* run-time test */
if (!_A_add_entry ("ADAMS_set", _A_uid_name, &_A_def, _A_CLASS, _A_TASK))
      {
      printf ("_A_add_entry failed for set ADAMS_set of ADAMS_class elements\n");
      _ADAMS_STATUS = 0;
      }
```

Subscript_Pool Declaration:

This feature is not implemented yet.

4.2.3. Instantiation Statements

Instantiations have some similarity to declarations in the viewpoint of the code generator. The main concern is in setting up and entering an *entry definition* in the dictionary for the instance. Most of the items in the *entry definition* of concern to declarations are not used in instantiations.

Attribute Instantiation:
Element Instantiation:
Map Instantiation:

These are all placed together because the parser does not distinguish among them. They are allowed to have one or more parent classes, and any number of association clauses. As an example, << **ADAMS_instance INSTANTIATES_A ADAMS_class1 and ADAMS_class2, scope is task** >> generates the following code:

```
_ADAMS_STATUS = 1;
                  /* declare element ADAMS_instance of   */
                  /* class ADAMS_class1 and ADAMS_class2 */
strcpy (_A_inst_uid, "3#");
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "ADAMS_class1", _A_USER);
                  /* handle argument # 1 */
                  /* name not found at parse time; find now */
strcpy (_A_uid_name, "");
_A_d_level =  _A_USER;
if ( _A_c_check ("ADAMS_class1", _A_uid_name, _A_CLASS,&_A_d_level) == 0)
      {
      printf ("_A_c_check failed: ADAMS_class1 not found\n");
      _ADAMS_STATUS = 0;
      }
else
      {
      strcpy (_A_args [1], _A_uid_name);
      }
_A_add_instclass (&_A_def, "ADAMS_class2", _A_USER);
                  /* handle argument # 2 */
                  /* name not found at parse time; find now */
strcpy (_A_uid_name, "");
_A_d_level =  _A_USER;
if ( _A_c_check ("ADAMS_class2", _A_uid_name, _A_CLASS,&_A_d_level) == 0)
      {
      printf ("_A_c_check failed: ADAMS_class2 not found\n");
      _ADAMS_STATUS = 0;
```

```
   }
else
      {
      strcpy (_A_args [2], _A_uid_name);
      }
                  /* run-time test */
if (!_A_add_entry ("ADAMS_instance", _A_inst_uid, &_A_def, _A_INST, _A_TASK))
      {
      printf ("_A_add_entry failed for ADAMS_instance s of class ADAMS_class1\n");
      _ADAMS_STATUS = 0;
      }
_A_uid_instant (_A_inst_uid, _A_args [1]);
_A_uid_instant (_A_inst_uid, _A_args [2]);
```

Set Instantiation:

This is very similar to the other three instantiations. The differences are: only one parent class is allowed, as it would not make sense to have more than one—what class of element would be allowed into such a set? ADAMS allows only one class of elements to be entered into any one set; a *initial_clause* is allowed here, which specifies the initial contents of the set.

Codomains/Subscript_pools:

ADD_CODOMAIN
EXTEND_SUBSCRIPT_POOL

Both the ADD_CODOMAIN and EXTEND_SUBSCRIPT_POOL statements are unimplemented as of yet. There are still semantic issues to be resolved regarding them.

4.2.4. ADAMS Variables

ADAMS_vars can be used in most place that regular ADAMS names are used. However, there are two statements that are directly concerned with ADAMS_vars: *VAR_decl* and *VAR_assign*.

VAR_decl:

If ADAMS_var declarations occur, they must be the first ADAMS statements in their functions, and must appear within or at the end of the C declarations in the function. The reason is that they are translated into C variable declarations, and would generate C errors if they occurred

elsewhere. The translation is to declare a C variable of name *_A_uid_*<*ADAMS_var name*>, of type *_A_uid_string*.

Example: << **adams_var x, y, z** >> is translated into:

```
_A_uid_string     _A_uid_x;
_A_uid_string     _A_uid_y;
_A_uid_string     _A_uid_z;
```

VAR_assign::

Code is generated to obtain the *uid* of the target ADAMS element, placing it into *_A_args* [0], then do a string copy from *_A_args* [0] into the adams_var name. This sounds like an obvious place for an optimization—after all, why place the *uid* into *_A_args [0]*, then copy into another string? The reason is that *_A_args [0]* obtains the *uid* through the general-purpose element handling routines, which have no way of knowing what has called them. Trying to make these element handling routines take care of all these special cases would make them totally unreadable and virtually unchangeable.

Example: << **x denotes ADAMS_element_inst** >> is translated into:

```
strcpy ( _A_args[0], <uid of ADAMS_element_inst> );
strcpy (_A_uid_x, _A_args [0]);
```

Notice that in this example of generated code we have used the expression <*uid of ADAMS_element_inst*>. Finding the *uid* associated with a particular ADAMS element designator is non-trivial. Often it requires several lines of C code. In this presentation, we have chosen to first illustrate the general patterns of code generated by individual ADAMS statements. In section 4.3, we will examine the issue of finding *uid*'s.

4.2.5. Set Manipulation Statements

The following is common to the generated code for most of the non-declarative ADAMS statements:

```
                    /* initialize _ADAMS_STATUS */
_ADAMS_STATUS = 1;
                    /* establish argument uid's */
strcpy ( _A_args[0], <uid of 1st argument> );
strcpy ( _A_args[1], <uid of 2nd argument> );
  .          .                    .
  .          .                    .
strcpy ( _A_args[k], <uid of k+1st argument> );
```

Since it will be applicable to most of the statements, it will not be repeated for each one; there will simply be a reference to which ADAMS element is which argument in the list, which will usually be clear anyway.

Most of the statements in this section are quite straightforward. The major work in each is in obtaining the *uid* for a set, perhaps also for an element, and then making the appropriate call to an index manager glue routine. As will be seen, the most difficult statements to handle are the union and intersection statements, mainly because they involve a variable number of arguments.

Make_empty:

Generate code to obtain the *uid* of the set, then call the Index Manager glue routine *_A_set_empty* ().

Example: << **make_empty ADAMS_set** >> is translated into:

```
strcpy ( _A_args[0], <uid of ADAMS_set> );
_A_set_empty ( _A_args[0] );
```

Insert and **Remove:**

Generate code to obtain *uids* of the respective element and set, and to call either the glue routine *_A_set_insert* () or *_A_set_remove* ().

Example: << **insert ADAMS_element into ADAMS_set** >> is translated into:

```
strcpy ( _A_args[0], <uid of ADAMS_element> );
strcpy ( _A_args[1], <uid of ADAMS_set> );
_A_set_insert ( _A_args[0], _A_args[1] );
```

Element removal, using **remove** is similar, with a call to *_A_set_remove* instead of *_A_set_insert*.

Set_copy:

Generate code to obtain *uids* of source and target sets, then code to loop thru source set, find each element there, and insert the element *uids* into the target set. The end result of this is two sets consisting of identical lists of *uids*.

Example: << **copy_to ADAMS_set1 from ADAMS_set2** >> is translated into:

```
strcpy ( _A_args[0], <uid of ADAMS_set1> );
strcpy ( _A_args[1], <uid of ADAMS_set2> );
_A_set_copy ( _A_args[0], _A_args[1] );
```

Complement:

Generate code to obtain the *uids* of the three sets involved. << set1 is complement of set2 wrt set3 >> means that set1 will contain all of those elements from set3 which are not already in set2. A call is then generated for the glue routine *_A_set_complement* ().

Example: << **ADAMS_set is_complement_of ADAMS_set2 wrt ADAMS_set3** >> is translated into:

```
strcpy ( _A_args[0], <uid of ADAMS_set> );
strcpy ( _A_args[1], <uid of ADAMS_set2> );
strcpy ( _A_args[2], <uid of ADAMS_set3> );
_A_set_complement ( _A_args[0], _A_args[1], _A_args[2] );
```

Intersection and **Union**:

The code generating functions for these two routines have passed to them a linked list of set names, as opposed to the normal single string for a set name in the other set statements. An array of *uids* is built up from this list, which along with *uid* of the target set, is passed onto either the glue routine *_A_set_intersect* () or *_A_set_union* ().

Example: << **ADAMS_set is_intersection_of ADAMS_set2, ADAMS_set3** >> is translated into:

```
strcpy ( _A_args[0], <uid of ADAMS_set> );
strcpy ( _A_args[1], <uid of ADAMS_set2> );
strcpy ( _A_args[2], <uid of ADAMS_set3> );
                    /* create null argument to end list */
strcpy ( _A_args[3], "");
_A_var_ptr = &_A_args[1];
_A_set_intersect ( _A_args[0], _A_var_ptr );
```

Union is virtually the same, merely substituting a call to *_A_set_union* in place of *_A_set_intersect*.

4.2.6. Looping

Looping is accomplished in ADAMS through the FOR_EACH stmt, which gives an ADAMS_var and a set name, for which each element has its *uid* placed in the ADAMS_var, one per iteration of the loop. The arbitrary nesting of FOR_EACH loops allowed in ADAMS presents some difficulties for the preprocessor in generating correct code. For every loop, a label of form _L<loop_num> is generated which is used for the << EXIT LOOP >> statement to jump to. The preprocessor must keep track of which loop is being ended when it encounters the end of one, not only for the label, but in generating a do-while construct. While the ADAMS user sees the syntax << **FOR_EACH x in ADAMS_element DO ...** >>, the preprocessor must generate more involved C code as shown below.

```
_ADAMS_STATUS = 1;
strcpy ( _A_args[1], <uid of ADAMS_element> );
{
_A_uid_string   _A_loop_uid1;
strcpy ( _A_loop_uid1, _A_args[1] );
if (_A_set_first_element (_A_uid_x, _A_loop_uid1) )
      do {
                    /* ADAMS/C code */
}  while (_A_set_next_element (_A_uid_x, _A_loop_uid1);
_L1:;
}
```

The code generated starts off with an initialization of _ADAMS_STATUS, then finds the *uid* of the set using the general element handling routines, placing it into _A_args [1]. _A_args [1] is then string copied into *_A_loop_uid%d*, where %d is replaced by the current_loop number. This number is unique for each *for_each* loop in the program. As the parser recognizes the start of a

new FOR_EACH loop, it makes a call to find a new loop number. As loops are nested, a linked list of these numbers is created, so that as loops are exited, it is always possible to obtain the correct number. Typical stack operations maintain this list. A new C block is created for each loop, right after the code is generated to obtain the *uid* of the set name. The main purpose of this block is to be to create the *_A_loop_uid%d* variables as needed, so as to avoid problems with having a fixed number of them declared in the header, and creating an error when the programmer creates one more loop than the preprocessor expected him to. The number of the loops is always increasing, as the numbers can not be reused, since the labels generated at the base of the loops must be unique. In a preprocessor for another host language that does not have the ability to create new blocks at will as in C, a fixed number of variables can be created, but this is clearly inferior to the current implementation. A conditional test is made on the value obtained from a call to the index manager glue routine *_A_first_set_element ()*, which obtains the *uid* of the "first" element in the set, placing it into the variable for the ADAMS_var (*_A_uid_x* in this case). (By "first" we mean that which the index manager sees as the first element, through its own algorithm for looping through the elements of a set. The ADAMS user can not count upon any particular ordering for the elements in a set, only that the loop will indeed give all of the elements, each one only once.) A do-while loop is started off, with the `do {` part generated, followed by the ADAMS/C code within the FOR_EACH loop body.

At the end of the loop the do-while construct is finished, with `} while (_A_set_next_element (...) );`, where *_A_set_next_element ( )* determines if there are more elements in the set left, and if so, places the *uid* of the "next" element into the adams_var variable. Finally, the unique label of the form _L%d is generated, followed by the close of the new C block.

The C code for a more complex example may be instructive:

```
<< FOR_EACH x in ADAMS_element_set1 do
        /* ADAMS (non-loop) / C code no. 1 */
        << FOR_EACH y in ADAMS_element_set2 do
                /* ADAMS (non-loop) / C code no. 2 */
        >>
>>
```

is translated into:

```
_ADAMS_STATUS = 1;
strcpy ( _A_args[1], <uid of ADAMS_element_set1> );
{
_A_uid_string     _A_loop_uid1;
strcpy ( _A_loop_uid1, _A_args[1] );
if (_A_set_first_element (_A_uid_x, _A_loop_uid1) )
      do    {
      /* translation of ADAMS (non-loop)/C code no. 1 */
_ADAMS_STATUS = 1;
strcpy ( _A_args[1], <uid of ADAMS_element_set2> );
{
_A_uid_string     _A_loop_uid2;
strcpy ( _A_loop_uid2, _A_args[1] );
if (_A_set_first_element (_A_uid_y, _A_loop_uid2) )
      do    {
      /* translation of ADAMS (non-loop)/C code no. 2 */
}     while (_A_set_next_element (_A_uid_y, _A_loop_uid2);
_L2:;
}

}     while (_A_set_next_element (_A_uid_x, _A_loop_uid1);
_L1:;
}
```

One note about the code for looping in ADAMS—generating this code differs from most of the other statements in that the code generation is not totally abstracted into one call at the statement recognition level in the parser. The code for obtaining the *uid* of the first element in the set must be generated when the first part of the loop ("FOR_EACH x in ADAMS_element_set") is recognized, while the code for the next_element function can not be generated until the base of the FOR_EACH loop is recognized. Since other ADAMS statements can occur in the interim, a single call to the code generation routines will not work in this case. This is the only case in ADAMS where this occurs, since the FOR_EACH statement is the only one which allows other ADAMS statements to be embedded within it.

4.2.7. Assignment Statements

There are two classes of assignment statements in ADAMS—1) those that make assignments between host and ADAMS variables—the FETCH and STORE statements; 2) those that make assignments between two ADAMS variables—the ATTRIBUTE_ASSIGN and MAP_ASSIGN statements.

Fetch:

The FETCH statement gets a value from an attribute of an ADAMS element, and stores it into a host variable.

The preprocessor calls the element handler to obtain the *uids* of the source element and attribute and stores them into *_A_temp_uid1* and *_A_temp_uid0* respectively (see section on element designators for details on why). A call is then made to *_A_attr_get_val ( )* using these uids, the host variable name, and an internal buffer size.

As an example, << **FETCH FROM x.attr1 INTO temp** >> results in:

```
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, temp, _A_FETCH_BUFF_SZ);
```

Store:

The STORE statement is the complement of the FETCH statement, taking a string from a host variable and storing it into an ADAMS attribute.

The uids are obtained as in the FETCH case. The function called is *_A_attr_insert ( )*. The example << **STORE FROM temp INTO x.attr1** >> results in:

```
_A_attr_insert (_A_temp_uid0, _A_temp_uid1, temp);
```

Neither the FETCH nor STORE statements currently handle ADAMS_vars in the place of the ADAMS element. This means that the following:

<< y DENOTES x.attr1 >>
<< STORE FROM temp INTO y >>

will not work, though it should be equivalent to the STORE example from above.

Attribute_Assign and **Map_Assign:**

The ATTRIBUTE_ASSIGN and MAP_ASSIGN statements are used to assign attributes and maps to ADAMS elements. The ADAMS elements may be either element designators or ADAMS_vars, but the ADAMS elements must be either both maps or both attributes.

Both of the statements are handled the same by the parser, since there is no syntactical method of distinguishing between them. Although the parser could have been modified to make the necessary dictionary lookups to distinguish between them, it was decided to keep the parser simpler by having the code generator make those calls.

There are a number of possible cases for the desination and source ADAMS elements:

(1) Both are ADAMS_vars—Handle just as for a VAR_ASSIGN statement. Strcpy the source uid into the destination ADAMS_var.

(2) One is an attribute element designator, the other is a map element designator. This is an error case.

(3) Both are attribute element designators. Obtain the uids of the objects and their attribute functions, then make call *_A_attribute_assign_value ( )*. An example is: << **x.attr1** = **y.attr2** >>, which results in:

```
_A_attr_insert ( _A_args[0], _A_args[1], _A_temp_uid0, _A_temp_uid1 );
```

(4) Both are map element designators. Obtain the uids of the objects and their map functions, then make call *_A_map_insert ( )*. An example is: << **x.map1** = **y.map1** >>, which results in:

```
_A_map_insert ( _A_args[0], _A_args[1], _A_args[2] );
```

(5) The destination is an ADAMS_var, the source is a map or attribute element designator. Obtain the uid of the element designator (If a map, evaluate to the target), and strcpy to the destination.

(6) The destination is an attribute element designator, the source is an ADAMS_var. This is similar to case 3, except that there is only 1 uid for the source, instead of two. We need a funtction to handle this case—not implemented at this time.

(7) The destination is a map element designator, the source is an ADAMS_var. This is similar to case 4, again with the exception that there is only one uid for the source, instead of two. However, this case can be handled by *_A_map_insert ( )* as well, since that function already expects the source element to be evaluated to its target, as opposed to remaining in a object/map function pair of uids.

4.2.8. Dictionary Manipulation Statements

The translation for these 3 types of ADAMS statements (delete, erase, rescope) is fairly simple, once the unique ids are obtained from the element handler. One item to note is that the dictionary call generated for each has a name and a unique id parameter, just as in most dictionary calls. If the name is non-null, then the unique id obtained during the call is placed back in the character buffer for the unique id parameter. If the name is null, then the dictionary utilizes the unique id parameter. Since we will have the unique ids available from the element handler, it was decided to make the name parameter a null, in order to ease problems with ensuring that the scope or "VAR " were properly removed form the ADAMS name if they were present.

For erase and rescope it is necessary to specify the kind of element that is being used (class, instance, codomain or subscript_pool). This allows the dictionary to expand the name space available for ADAMS elements.

Delete::

For the example << **delete ADAMS_element** >> the following code is generated:

```
if(!_A_c_delete_entry ("", _A_args[0], _A_INST, &_A_d_level));
      {
      printf ("_A_c_delete_entry failed for ADAMS_element0);
      _ADAMS_STATUS = 0;
      }
```

Erase::

For the example << **ERASE codomain ADAMS_codomain** >> the following code is generated:

```
if(!_A_c_erase_entry ("", _A_args[0], _A_CO, &_A_d_level));
      {
      printf ("_A_c_erase_entry failed for ADAMS_element0);
      _ADAMS_STATUS = 0;
      }
```

Rescope::

For the example << **RESCOPE CODOMAIN ADAMS_codomain AS TASK** >> the following code is generated:

```
if (!_A_rescope ("x", _A_args[1], _A_CO, _A_LOCAL, _A_SYSTEM))
      {
      printf ("_A_c_rescope failed for ADAMS_element0);
      _ADAMS_STATUS = 0;
      }
```

There are limitations on the rescope command which the ADAMS user should be aware of, regarding the direction of allowable rescoping, and of constituent element scope levels. Only upward rescoping is allowed, that an entity can only have a greater scope after rescoping than before. The purpose of this is to ensure that no entry that is dependent upon an element being of *system* or *task* level fails due to that element being rescoped to a lower level that is out of scope of the entry. One should be able to make tests for such dependencies, and then allow the rescope only if it will be safe, but that is the manner in which the dictionary is currently implemented.

An upward rescoping is allowed for classes only if all the constituent elements (maps and attributes) are of at least the new scope level. A class which has *user* scope and has maps and attributes all of *user* level cannot be rescoped to either *task* or *system* until all of the maps and

attributes have been rescoped to that level.

4.3. General code generation for element handling

An important part of the code generated for almost every ADAMS statement is that code for obtaining a unique id for an element. In the preceding examples, this has been subsumed by the generic expression <*uid for ADAMS_element*>. Since there are numerous cases to be examined for each ADAMS element, it is desirable to abstract all of this testing out of the statement level, and consider it at the element handling level.

There are two methods of accessing this level, one for a fixed number of elements in a statement, and another for a variable number. Two statements need the variable case—intersection and union statements, since the syntax fixes no bound on the number source sets for these operations. For the rest of the statements, a call to function *handle_element* ( ) generates the necessary code for obtaining a unique id. The variable cases require a call to *var_args ( )*, which takes an array of ADAMS elements and loops through them, calling *handle_element* ( ) for each one. Figure 4-1 will help to illustrate this and other facets of obtaining *uids*.

We now examine the various cases for generating code to obtain the *uid* of an element. Those cases are:

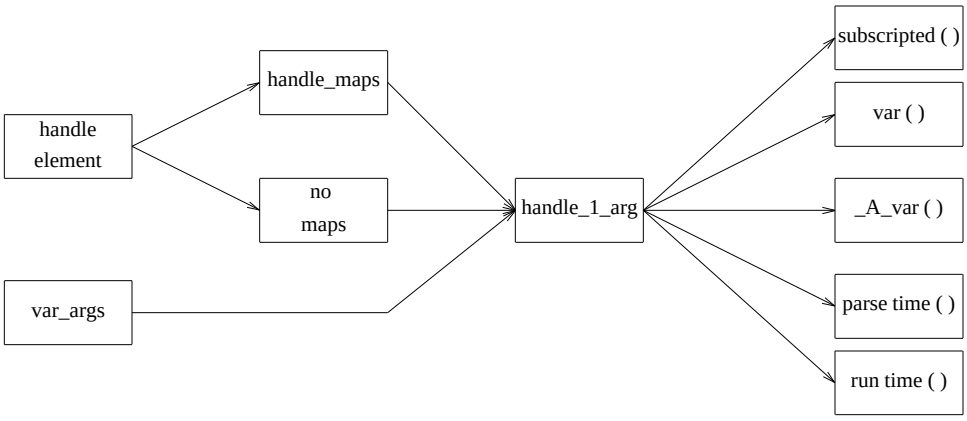


Figure 4-1

- scoped names
  - subscripted variables
  - map/attribute in name
  - ADAMS_vars
  - var variables
  - generic literal name

4.3.1. Scoped Names

All named ADAMS elements can be scoped directly through their names, by placing the scope level and whitespace before the actual names, as in:

<< REMOVE system ADAMS_element from ADAMS_element_set >>.

The effect of this is that if there are several ADAMS elements in the dictionary in the available name space of the user, but at different levels, then the particular one desired is accessed, as opposed to going through the default path (ie, LOCAL, then USER, TASK, and SYSTEM). The parser is able to recognize the combination of (scope_level actual_name) as an allowable name. Instead of passing just the actual_name as a string to the code generator, it passes the concatenation of the scope level (in capital letters), a space, and the actual_name. In the example above, the parser would pass "SYSTEM ADAMS_element" as the parameter for the name.

Checking for the name being scoped is the first thing that it does with an ADAMS name. If one is found, the *elem_scope* variable is set to reflect the scope found. If none is found, then it is set to the default scope, which is dependent upon whether the element is being used in a declaration statement or not. If it is in a declaration, the default scope will most likely be USER, as it makes little sense to have components of a persistent declaration be of temporary scope. For element references outside of declarations, it is perfectly fine to be of LOCAL scope, and therefore the dictionary should start its search at the LOCAL level.

The *handle_element* ( ) routine calls a function that checks for the scoped name, and if this is successful, it then calls a routine to strip the scope level away from the rest of the name, since the scope value has been captured in the scope variable.

4.3.2. Subscripted Names

Subscripted names are supported in ADAMS, with many of the traditional features of arrays, but also with some important differences. An ADAMS array is a set of named ADAMS elements with a common base name, and each of which has a unique set of subscripts, which is used to access the particular element. This serves to extend the ADAMS name space in a controllable manner.

The sets of subscripts consist of names from user-defined *subscript pools*, which are enumerations of allowable elements in the domain for a subscript index. There is a predefined subscript pool for the non-negative integers. These pools are extensible, there being a method for adding elements to (though none for removing them from) a subscript pool.

The elements of a subscripted variable are not stored in the traditional array method, that is, contiguously. There would be little point to storing all the elements of an ADAMS subscripted variable together in one location, as this would likely defeat the benefits of having ADAMS running on parallel machines. There must be methods allowed for data migration on such machines; forcing all the elements together would prevent this.

The lack of contiguous storage and of the dynamic nature of the subscript pools disallows the usual method of calculating offsets into an allocated storage to find the location of a particular array element. The standard formulas for calculating offsets won't work with data that is broken up or moved around. A method that solves this problem must still contend with the dynamic size problem. For example, a mapping that is valid for an array with bounds of [10, 25, 30] must still be valid when the dimensions are extended, to say [10, 30, 35].

The ADAMS method of handling arrays is as follows. The basename of the array must be entered into the dictionary along with information on the number of subscripts and the subscript pools they are obtained from. When an element is referenced, the set of subscripts for an element are mapped onto an integer that is unique for all possible elements in the array (both the existing

elements and those that could be instantiated once the bounds are increased). The uniqueid manager is given the *uid* of the basename and this unique integer, and performs its own mapping to determine the uniqueid of the individual element. Once this is obtained, the array element is treated just like a non-subscripted element.

The problem arises of how to calculate these unique integers. The method must guarantee that no two elements in a particular array are mapped onto the same integer, while ensuring that the unique integers obtained are within the representable bounds of the system. For example, an easy method to calculate unique integers would be to use a product of prime numbers as follows: For each of the *n* subscripts in identifier [i_1, i_2, ... i_n], raise the *k*th prime number by $index_k$. Multiplying all of these terms together gives a unique integer, $2^{i_1}\ 3^{i_2}\ \cdots\ prime_n^{\ i_n}$ assured by the fundamental theorem of arithmetic. The numbers generated by this method would quite quickly exceed the abilities of most machines to store them. Foo [32, 10, 10] would give a unique number around 2.4767×10^{21} —Subscripts in the hundreds would never be handled by most machines.

The method used in ADAMS lessens this problem of obtaining extremely large integers. A geometric model is used to derive these integers. For simplicity we will examine a 3-dimensional array that uses only the natural numbers for subscripts. It will be seen that this method is applicable to any k-dimensional array, (k >= 0), and to any subscript pools. By "a tetrahedron of size n", it is meant a tetrahedron bounded by the k axis, and the lines which intersect those axis at value of n. Figure 4-1 shows 3-dimensional tetrahedrons of sizes 0, 1, and 2.

For a particular element of the array, sum the 3 subscripts, which we will refer to as $i_{total} = \sum_{k=1}^{n} i_k$. Now imagine a 3-dimensional tetrahedron, which is bounded by the points ($i_{total,}$ 0, 0), (0, $i_{total,}$ 0), and (0, 0, i_{total}). The three tetrahedrons in Figure 4-1 are formed by $i_{total} = 0$, 1, and 2 respectively. Examining the figure it should be readily apparent that the 3-dimensional tetrahedron of size i_{total} can be broken down into a 3-dimensional tetrahedron of size

$\left\{ i_{total} - 1 \right\}$ and a set of points that consist of the outermost points in the tetrahedron, which can be projected onto a 2-dimensional surface of size i_{total}. The tetrahedron of size 2 in Figure 4-2 can be broken down into one of size 1 and the plane of points that form the differential between a tetrahedron of size 1 and 2. The points in this plane are delimited by circles, while those in the tetrahedron of size 1 are delimited by squares. Figure 4-3 shows this projection of the outermost slab onto the x-y axis.

This leads to an idea for numbering the points. Assume that there are N points in the $i_{total} - 1$ length tetrahedron , and M in the new slab, and that the N points are numbered from 0 to $N - 1$, ignoring the details of how they got numbered for now. If one can assure a scheme whereby those M points in the new slab get numbered (in some manner) from N to ($N + M - 1$), and that all points in any new slabs added will be higher than (N + M - 1), then the problem is reduced by one dimension.

In the example, let us try to number the points in the outermost slab. By inspection, we can see that there are 4 points in the next smallest tetrahedron, and 6 points in the new slab. So the first 4 points will be numbered from 0 to 3, and the points in the slab from 4 to 9. Now project

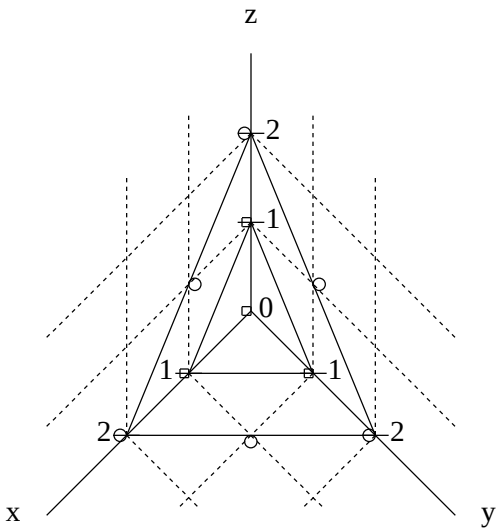


Figure 4-2.

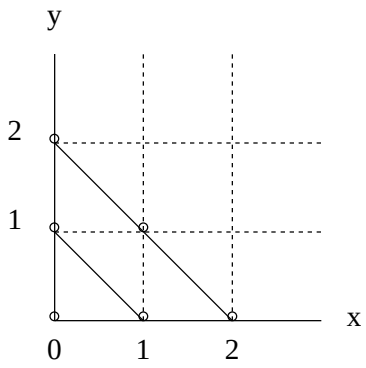


Figure 4-3.

those 6 points onto the plane, as in Figure 4-3. The equivalent of the slabs from the 3-dimensional case are now lines in 2-dimensions. Take a point in this plane, say (2, 0), and use the same method as in the 3-dimensional case. One possible difference could be that the point would not in the outermost slab anymore, in which case we would focus on the smaller tetrahedron that it defines, which is the one in which the desired point is in the outermost slab. That does not occur in this example though. The number of points in tetrahedrons smaller than the outermost one is 3. We are left with the slab consisting of the points (0, 2), (1, 1) and (2, 0). Now we can project those points onto the x-axis, which gives a unique number for each point, eliminating the need for continuing with the algorithm. The number assigned to the point (2,0,0) (which is the location of this point in the 3-dimensional tetrahedron) is equal to the sizes of the smaller tetrahedrons (both the 3 and 2 dimensional ones) and the new offset from the x-axis, 0. This comes to 4 + 3 + 2 = 9.

Therefore, the general formula for calculating the unique integer for an array element with subscripts $i_{1,} i_{2,} \cdots, i_k$ is: $f\,(i_{1,}\ i_{2,}\ ...,\ i_k\,) = \mathit{sizeof}\,(k,\ n-1) + f\,(\,i_{1,}\ i_{2,}\ ...,\ i_{k-1})$, $n = \sum_{j=1}^{k} i_j$, $f\,(\,i_1\,) = i_1$. The problem now becomes how to determine the number of elements in a tetrahedron of arbitrary size and dimension. We will first show that the number of elements in a 2-dimensional tetrahedron of size n (range of values is from 0 to n) is : $f\,(\,n\,) = \frac{(\,n+1\,)\,(\,n+2)}{2.}$ and then extend the formula to k-dimensions.

Assume this is so for $f\,(n-1)$. Now determine f (n). f (n) = f (n - 1) + points in new slab. It can be seen that there are (n + 1) points in the new 2-dimensional slab, as they can be projected onto the x-axis from 0 to n. So

$$f\,(n) = f\,(n-1) + (n+1) \;=\; \frac{(n-1+1)(n-1+2)}{2} + n + 1 \;=\; \frac{(n+1)(n+2)}{2.}$$

The formula works for n = 0, giving 1 as expected. Using the induction step, this assures that the formula is indeed correct. The principle of induction assures that this formula is true for

all n ≥ 0. Therefore the *sizeof* ( ) function $f(2, n) = \frac{(n+1)(n+2)}{2}$. This can also be expressed as $\left\lceil \begin{matrix} n+2 \\ 2 \end{matrix} \right\rceil$.

We can break a k-dimensional tetrahedron of size n into n tetrahedrons of dimension (k - 1), by recursively extending the prior argument about breaking down a k-dimensional tetrahedron of size n into a k-dimensional tetrahedron of size (n -1) and a (k-1) dimensional tetrahedron of size n. So to find the number of elements in a k-dimensional tetrahedron of size n, all one must do is sum up the elements in (k - 1) dimensional tetrahedrons ranging from size 0 to n. In [GKP89] it is shown that $\sum_{k=0}^{n} \left\lceil \begin{matrix} r+k \\ k \end{matrix} \right\rceil = \left\lceil \begin{matrix} r+n+1 \\ n \end{matrix} \right\rceil$. The formula for f (2, k) is already in the proper form for this summation. Therefore the formula for f (3, n) = $\left\lceil \begin{matrix} n+3 \\ 3 \end{matrix} \right\rceil$, and the general formula is:

sizeof (k, n) = $\left\lceil \begin{matrix} n+k+1 \\ k \end{matrix} \right\rceil$.

The formula for calculating the unique integer for an array element with subscripts $i_1, i_2, \cdots, i_k$ is therefore:

$$f(i_1, i_2, \ldots, i_k) = \mathit{sizeof}(k, n-1) + f(i_1, i_2, \ldots, i_{k-1}) ,$$

which reduces to:

$$f(i_1, i_2, \ldots, i_k) = \left\lceil \begin{matrix} k-1+\sum_{l=1}^{k} i_l \\ k \end{matrix} \right\rceil + f(i_1, i_2, \ldots, i_{k-1}) ,$$

$$f(i_1) = i_1 .$$

Solving the recurrence gives a final solution of:

$$f(i_1, i_2, \ldots, i_k) = \sum_{j=1}^{k} \left\lceil \begin{matrix} j-1+\sum_{l=1}^{j} i_l \\ j \end{matrix} \right\rceil .$$

Subscripted variables—Code generation

There are two separate times at which code is generated which involves subscripted names: first at the time of instantiation of the array; second at the time an individual element is referenced, which may include instantiating that individual element.

—Subscripted variable instantiation

A subscripted name instantiation from the example program is:

<< ADAMS_element [integer, integer] instantiates_a FOO_CLASS >>

This differs greatly from the typical array instantiation in traditional programming languages. The norm would be to allocate a contiguous amount of memory of the size of the total number of elements in the ranges of the subscripts, each of which is a fixed size at instantiation time.

```
int   ADAMS_element [10][20];
```

is a typical example, allocating 200 elements, with respective ranges of 10 and 20 in the two dimensions. There is generally a simple formula for determining the offset of a particular element into this chunk of memory.

In ADAMS, *subscript pools* can be used in place of the integer ranges. A subscript pool is a set of allowable elements to be used for a subscript index. A pool can have elements added to it, so the range is never fixed as it is in traditional programming languages. An instantiation of a subscripted name in ADAMS simply enters the name of the array into the dictionary, along with a unique id for the whole array, the number of subscripts needed to access individual elements, and the subscript pools that each of the indices are to be obtained from (they need not all be from the same subscript pool).

—Subscripted variable element references

The code that is necessary for referencing subscripted names is built upon that for referencing unsubscripted name, as the first step is to get the *uid* for the base name. A unique integer index is obtained for this element, using the invertible function f that maps all uniquely subscripted tuples onto a non-negative integer, and both of these are used to finally obtain the *uid* of the specific element.

Looking at the following line from the sample program will help to illustrate this:

<< DELETE ADAMS_element [33, 5] from FOO_SET >>

The steps that the preprocessor goes through to obtain the *uid* of a subscripted element are as follows:

—Obtain the basename of the subscripted element.

—Use this name in a call to the dictionary to obtain its *uid*, just as though it were not a subscripted name.

—From the rest of the element name, retrieve a list of the subscripts.

—Query the dictionary for the integer mapping for each of the subscripts. If the subscript is not obtained from the INTEGER subpool, then the dictionary maintains a mapping from the elements of the domain of a subscript pool onto the natural numbers. The dictionary is given the array basename, and at the time the array was instantiated the dictionary knew how many subscripts the array had, and the subscript pool from which its values are obtained. It is thus a simple matter for the dictionary to map the set of subscripts onto a set of integers.

—Calculate a unique integer for the set of subscripts, using the algorithm in section yy.

—Call the index manager with the *uid* of the basename and the unique integer. If the element already exists then the *uid* of the element is returned. If it doesn't exist, a null *uid* is returned. In this case, a call to *_A_unique_getuid ()* is needed to obtain a new *uid*, followed by a call to the index manager again, informing it of the basename *uid*, unique integer, and the *uid* of the element

to associate with the two.

Note that the general ADAMS philosophy is to instantiate a subscripted variable element when it is first referenced. But at times this might not make much sense, as in the above delete example. If *ADAMS_element*[33, 5] does not already exist at the time that ADAMS statement is reached, then what is the point to instantiating that element, as there will immediately will be an error anyway, since although the element is created, it cannot possibly be a member of FOO_SET. The option exists to change this at the statement level, as the calls described here are all lower-level than the ADAMS statement level, and the code generator could be readily changed to reflect a differing philosophy if it is so desired.

4.3.3. General Element Designators

An element designator is the name of an element that is obtained by first finding a base element, then following a (possibly null) list of maps, and ending with a final map or attribute. For example, *x.map1.map2.attr1* and *y.attr2* are element designators.

Finding the *uid* of an element designator is more complicated than doing so for a regular element. This is because a chain of map/attribute links must be traversed, obtaining a new *uid* at each step. A regular element is denoted by a single *uid*. An element designator can be denoted by the *uid* of the final target, or by an object/function *uid* pair. For example, by *x.map1* it is meant that there is an object *x* that has a map denoted by *map1*, which in turn has some image, which we will call the *target*. When evaluating *x.map1*, one can obtain the *uid* of the *target*, or simply refer to the *uids* of the object x and the map map1, which as a pair uniquely identify the *target*. There are some calls to the index manager which require one format and some which require the other. Some require both, for different elements. For example, in a MAP_ASSIGN statement such as **<< x.map1 = y.map1 >>,** the index manager routine *_A_map_insert ( )* needs the separate *uids* of x and map1 for the *destination*, and the *uid* of the *source* y.map1.

To obtain the *uid* of the target, one starts with a list of the component names in the element designator. The *uid* of the first named element is placed into temporary *uid* storage _A_temp_uid1. The *uid* of the second is placed into _A_temp_uid0. If there is another link after this, then the function *_A_map_get_val ( )* is called, and places the *uid* of the target into _A_map_target. This is now the new element, which has the map function which follows in the name. For example, in *x.map_a.map_b* we first view this ADAMS entity as an object *x* which has a map *map_a* that starts a link which will lead to the eventual *target*. After obtaining the *uids* for *x* and *map_a*, and using these to evaluate *_A_map_get_val ( )*, we have a new entity which has a *map_b* which will lead to the *target*. The *uid* of this new entity is copied into *_A_temp_uid1*, since this location retains the *uid* of the object. In this case, the chain ends after *map_b*, and the value of the *uid* of *map_b* stays in *_A_temp_uid0*.

The function that desired the *uid* of the element designator passes a boolean parameter that indicates whether or not to evaluate the final link to the target and obtain its *uid*, or to leave the *uids* of the final object and map in *_A_temp_uid1* and *_A_temp_0*.

4.3.4. ADAMS_vars

Unlike the other special cases in this section, an ADAMS_var has no particular distinguishing syntactical feature when it is referenced. The only way to detect usage of an ADAMS_var is to keep keep track of those which are declared in the << **ADAMS_var [name_list]** >> statement, and check each name against that list. In some sense, this is performing the type of operation that the dictionary does for regular named elements; however the dictionary cannot be utilized for this function, because the ADAMS_var elements do no not have persistent names, merely persistent *uids*. The ADAMS_var name has scope only for the duration of the ADAMS program, and as far as the ADAMS run time system is concerned, it is a more indirect method of referring to an element. Referring to an element by name is already an indirect method, since there must be a mapping from name onto the unique id. The name however, does refer to one particular unique ele-

ment; the unique id id to which it maps does not change, unless one deletes the named element and then instantiates it again. With ADAMS_vars though, there is not the same sense of a relationship between the name and a unique id. For a period, the ADAMS_var will indeed refer to a particular unique id. However, since one of the major reasons for having ADAMS_vars is to provide for mechanism for handling variables within the **FOR_EACH** loops, the same ADAMS_var will generally refer to a number of different ADAMS elements over the duration of the ADAMS program.

4.3.5. Var Variables

A var variable is easy to recognize, as the user must prepend the word "var " (any mix of upper/lower letters) to the start of the name, as in << **DELETE var ADAMS_element** >>. The parser recognizes that as a var variable, and sends "VAR ADAMS_element" to the code generator as the element name. All that is necessary for the code generator to do to detect it being a var variable is to pass the name to a function that checks the first four characters for a match with "VAR ".

If no such match is found then nothing else is done in regards to var variables for that name. If the match is made, then code is generated which is similar to that for regular ADAMS names, except that the names are not quoted as they are for the literal ADAMS names. In addition, code must be generated to make a check at run time on the name, to see if is scoped at that time. In this version of ADAMS, var variables in programs are not allowed to be scoped (ie, it is illegal to say "<< **DELETE system var ADAMS_element** >>", but the name at run time can be scoped—it is legal to specify "<< **DELETE var ADAMS_element** >>", and then at run time have the variable ADAMS_element take on the value "system xyz", so that it is the ADAMS element with name *xyz*, at level SYSTEM that gets deleted.

4.3.6. Generic Literal Names

The only way to determine that a name falls into this category is to make all of the tests for the special cases, and see that they fail. When this point is reached, a call to the dictionary routine *_A_c_check ( )* will obtain the unique id if the name has already been used. A discussion in section 5 centers around whether or not it should be allowable to handle a name which has not already been declared at parse time, under the assumption that it could be created by run time, with provisions for error handling if it is not found then either.

Every time the element handler is called, an argument number is passed to it. This is used in determining where the unique id is to be stored. For example, an argument number of 1 will lead to the unique id being stored in *_A_args* [1].

5. ADAMS-related issues

5.1. Embedded language issues

The design of ADAMS as a "common interface language" which is embedded in multiple host languages is a fundamental design decision. An alternative approach is to create a complete new language—often by extending an existing one, as was done in EXODUS [CDV88] with its E language, which is an extension of C++. There are advantages and disadvantages to the ADAMS approach.

The overriding reason that this approach was taken was so that people who use different programming languages could have a uniform way to create and maintain persistent databases, while still using a language familiar to them. Programmers can still work in their favorite host language, be it FORTRAN, C, Pascal or whatever. A major problem with traditional database languages is that they are totally foreign to the intended users, leading to great resistance for learning and actually using them. The ADAMS approach avoids this problem that other systems, such as EXODUS, suffer from. Nobody needs to learn E, C++, or any totally unfamiliar language to handle ADAMS.

The drawback to ADAMS's embeddability is that a programmer is no longer using a "clean" language; there are now aspects of two languages that he must be able to handle. There must be two different modes of thinking: ADAMS and host language modes. There is also the difficulty in ensuring that there is a proper interaction between the host language and ADAMS.

5.2. Parse time vs. run time checks

The persistent scope of names in ADAMS creates problems that are not present in normal programming languages. In non-persistent languages actual variable names can be replaced by appropriate offsets into memory by the time the compile/link phases are over. The actual variable names are not needed at run time unless using a debugger, but in either case the names go

away when the program ends.

The situation is entirely different when it comes to the persistent name space of ADAMS. Dealing with names in ADAMS is analogous to dealing with files in UNIX, where there is the concern of whether or not a file you are writing to already exists, and where users with privileges to write to group or system level files can remove or modify files that a number of users see, and in fact may already be depending upon. With ADAMS there is the additional factor of a time lag in between compile time and run time. The elements in the name space may have changed drastically during that period, with possible effects on any program that use the changed elements.

There are several ways in which entities may change. First, an element or class may exist at parse time, but is subsequentially deleted before the program is executed. This obviously would lead to errors, analogous to attempting to perform operations on a file that has been deleted. A possible solution would lie in the reference counts that are kept on elements; increase reference counts as elements are referenced at compile time. However, allowing reference counts to be created as a result of programs that have only been compiled and not run yet could be disastrous. Presumably reference counts would not be incremented at run time if they were already increased at compile time. If the program is never run that potential user of the entity will never have the opportunity to delete the element (i.e., remove his reference). This could lead to having many ADAMS elements be unable to be deleted, if one of the programs that increased the reference count were deleted before it was ever run. There is also the problem of what to do when a program is recompiled before it is ever executed. Link counts would be increased drastically if they were increased with every compilation. Deletion of persistent elements is a difficult enough problem without artificially making it even worse.

A second change occurs when an element that did not exist at parse time is created before the program is run. There are several options in this case. One could require compile time existence of all entities and generate fatal compiler errors otherwise. This would require recompi-

lation of the program after the entity is created. This could be unduly restrictive, as a user could create one program to define new classes and instantiate elements, and another to use these entities. Another option is to do as was done in the preprocessor, generate code that will perform run-time checks on the desired entities.

A third possible change occurs if an entity is modified between parse time and compile time. By modification we mean the class (or type) of the entity changes. In some regards this is similar to the first case, as the entity at run time is not what the preprocessor saw and generated code for. Some mechanism is needed to keep track of whether or not a class definition has been modified (for example, by obtaining new maps and/or attributes.) The time of modification would be very important, as simply knowing that a change has occurred is not sufficient; it must be determined whether that modification occurred before or after compilation of the program, since the modification is irrelevant if the modified version was the one seen at parse time.

It is not all clear what should be done in each of these cases; this is an open research area. One can take the conservative position that all entities must be defined at parse time and cannot change before run time without inducing an error. On the other hand, one can have more flexibility and allow entities to be created or modified between parse and run times, if there are appropriate checks to ensure the integrity of the entities.

One problem that none of these address is that of the structure of ADAMS elements being structurally modified or deleted by other users during execution of a program. With the above schemes it is possible to determine if the elements referenced at parse time still exist at run time. Nothing is done about the elements being modified (not deleted) prior to program execution. Using some fields with each element to test for date of last modification would take care of that problem, but would not help the run time modifications. If the test is done only at the start of the program the program has no way of ensuring the integrity of the ADAMS elements it deals with. It may well be necessary to do run time checks on the unique id (and possible associated fields)

for all element references. This will of course be expensive and as such is undesirable.

6. Conclusions

This report has presented the basics of understanding what was involved in developing the preprocessor for ADAMS embedded within C. It should serve as a good starting point for someone trying to develop an ADAMS preprocessor for other host languages.

There are several areas of work for future ADAMS preprocessors:

• Embedding ADAMS within other host languages is very desirable—ADA, FORTRAN and Pascal are logical languages to be handled next.

• Have ADAMS fulfill its original goal, that of a database implementation tool that runs on and takes advantage of parallel processors—the current version is strictly serial. Since very little in the current language is strictly geared for parallelism, the implementation must handle that behind the scenes. It would of course be possible to make changes to the ADAMS language itself, in order to explicitly incorporate parallel constructs.

• Looking into the possibility of making ADAMS an extensible language, like C++, eliminating the need for delimiters and enabling a parser to take advantage of information available from the "host" language.

• Implementing features of the language that are not fully, or at all, implemented as of yet: parameterized names, inverse retrieval operations, codomain methods, transactions. While the parser recognizes these constructs, nothing is done in the code generation phase, mainly due to debate over the exact semantics that should be involved.

Part of the problem in implementing features such as these is that the development of the preprocessor reveals flaws in the planned semantics. Lexical and parsing limitations force compromises in the language. Debugging efforts have revealed omissions in the language that had been overlooked. The development of semantics of ADAMS and the implementation have most definitely not been cleanly separated as a textbook model might suggest. Certainly one should have a very good of the desired semantics before developing a system such as this, else

the final results will be series of kludges, precisely what ADAMS has tried to avoid. However one must have the sense to realize that what seems sound on paper will need revisions, and be willing to change the plan as one gains experience with the concepts involved.

7. References

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

[GKP89] R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1989.

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

[Joh78] S. C. Johnson, A Portable Compiler: Theory and Practice, *Proc. 5th ACM Symp. on Prin. of Prog. Lang.*, Jan. 1978, 97-104.

[KeP84] B. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice Hall, Englewood Cliffs, NJ, 1984.

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

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

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

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

Appendix 1: ADAMS Preprocessor Options.

The following are the options that the ADAMS preprocessor recognizes:

-u *user_id*

Specify *user_id* to be used for dictionary access. The default is that *getuid ( )* call is made in the main program to determine the users UNIX id.

-t *task_id*

Specify the *task_id* to be used for dictionary access. The default is TASK_DEFAULT, a constant defined in a header file.

-d*pathname*

Specify the pathname to be used to find the top-level of the dictionary. Can be specified as an argument to the preprocessor through this option, or by setting the shell variable DICTPATH to the desired path.

-p

Generates comments in the generated C file that enclose the original ADAMS code. The comments are placed *after* the code that is generated for the ADAMS statements. This option can be useful for debugging.

Appendix 2: BNF for ADAMS language

The following is the BNF for the ADAMS language used in this version of the preprocessor.

<ADAMS_stmt> ::= <b_delimiter> <statement_body> <e_delimiter>

<b_delimiter> ::= <<

<e_delimiter> ::= >>

<statement_body> ::= <open_ADAMS_stmt>
<codomain_decl_stmt>
<subscript_pool__decl_stmt>
<extend_pool_stmt>
<add_codomain_method>
<attribute_decl_stmt>

<attribute_instance_stmt>
<map_decl_stmt>
<map_instance_stmt>
<class_decl_stmt>
<elem_instance_stmt>

<delete_element_stmt>
<variable_decl_stmt>
<set_decl_stmt>
<set_instance_stmt>
<view_stmt>

<fetch_stmt>
<store_stmt>
<looping_stmt>
<end_loop_stmt>
<set_copy_stmt>

<set_assign_stmt>
<make_empty_stmt>
<insert_stmt>
<remove_stmt>
<union_stmt>

<intersect_stmt>
<complement_stmt>
<var_assign_stmt>
<rescope_stmt>
<erase_entry_stmt>

<start_trans_stmt>
<abort_trans_stmt>
<end_trans_stmt>
<lock_stmt>
<unlock_stmt>
<close_ADAMS_stmt>

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

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

Codomain Syntax

<codomain_decl_stmt> ::= <codomain_name> **isa** CODOMAIN
<membership_clause>
[<access_method_clause >]
[<other_method>]
[<undefined_clause>]
[<unknown_clause>]
[<scope_clause>]

<codomain_name> ::= <actual_name>

<membership_clause> ::= **consisting of** #<regular_expression># |
validated by <codomain_method_def>

<access_method_clause> ::= fetch: <codomain_method_def>
store: <codomain_method_def>

<other_codomain_method> ::= <method_name>: <codomain_method_definition>

<method_name> ::= <actual_name>

<undefined_clause> ::= **udf** = <literal_value>

<unknown_clause> ::= **ukn** = <literal_value>

<literal_value> ::= ’ <codomain_value> ’

<codomain_method_def> ::= <extern_def_codomain_method> |
<locally_def_codomain_method>

<extern_def_codomain_method> ::= **EXTERNAL** <name>

<locally_def_codomain_method> ::= <host_language_proc>

<subscript_pool__decl_stmt> ::= <subscript_pool_name> **instantiates_a SUBSCRIPT POOL**
of <codomain_name> **values**
[<consisting_of_clause>]

<extend_pool_stmt> ::= **add** <subscript_value> **to** <subscript_pool_name> **POOL**

<subscript_pool_name> ::= <actual_name>

<add_codomain_method> ::= **add method to** <codomain_name> CODOMAIN
<method_name>: <codomain_method_def>

Attribute Syntax

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

<attr_class_entry> ::= <dict_class_entry>

<attribute_instance_stmt> ::= [**var**] <attr_entry> **instantiates_a** <attr_class>
[<scope_clause>] |
<ADAMS_var> **instantiates_a** <attr_class>
[<scope_clause>]

<attr_entry> ::= <dict_instance_entry>

<fetch_stmt> ::= **fetch into** <host_variable> **from** <value_desig>

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

<attr_assign_stmt> ::= **assign into** <element_desig>.<attr_desig> **from** <value_desig>

<value_desig> ::= <element_desig>.<attr_desig> |
<literal_value>

Map Syntax

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

<map_class_entry> ::= <dict_class_entry>

<map_instance_stmt> ::= [**var**] <map_entry> **instantiates_a** <map_class>
[<scope_clause>] |
<ADAMS_var> **instantiates_a** <map_class>
[<scope_clause>]

<map_entry> ::= <dict_inst_entry>

Class Syntax

Classes and Instances

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

<elem_instance_stmt> ::= [**var**] <dict_inst_entry> **instantiates_a** <class_name>
[**AND** <class_name>]*
[<scope_clause>] |
<ADAMS_var> **instantiates_a** <class_name>
[**AND** <class_name>]*
[<scope_clause>]

<super_class> ::= <dict_class_entry> [**AND** <dict_class_entry>]*

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

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

<synonym> ::= <actual_name>

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

<clustered_attr_enum> ::= ’{’ ’(’ <attr_cluster> ’)’ [, <attr_cluster>]* ’}’

<attr_cluster> ::= (<attr_cluster>) |
<attr_cluster>, <attr_cluster> |
<enumeration_element>

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

<delete_element_stmt> ::= **delete** <element_desig>

Class Syntax

Predicates

<predicate> ::= <disjunct> [**or** <disjunct>]*

<disjunct> ::= <conjunct> [**and** <conjunct>]*

<conjunct> ::= <term> | (<predicate>) |
<quantifier> ’[’ <predicate> ’]’

<term> ::= <equality_comparison> | <order_comparison>

<equality_comparison> ::= <element> <equality_test> <element> |
<data_value> <equality_test> <data_value>

<order_comparison> ::= <data_value> <order_test> <data_value>

<element> ::= <logical_var> | <element_desig>

<data_value> ::= <literal_value> | <element>.<attr_desig>

<equality_test> ::= = | !=

<order_test> ::= < | <= | > | >=

<logical_var> ::= <bound_var> | <free_var>

<quantifier> ::= (**all** <bound_var> **in** <set_desig>) |
(**exists** <bound_var> **in** <set_desig>)

<free_var> ::= $X | $x

Set Syntax

Denotation

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

<set_class_entry> ::= <dict_class_entry>

<set_instance_stmt> ::= [**var**]<set_entry> **instantiates_a** <set_class>
[<initial_clause>]
[<scope_clause>] |
<ADAMS_var> **instantiates_a** <set_class>
[<initial_clause>]
[<scope_clause>]

<set_class> ::= <class_name>

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

<view_stmt> ::= <set_desig> **attributes_of** <class_name> |
<set_desig> **maps_of** <class_name>

Set Syntax

Manipulation

<looping_stmt> ::= **for_each** <ADAMS_var> **in** <set_desig> **do**
[<host_language_statement>]*
[<ADAMS_statement>]*

<end_loop_stmt> ::= **exit_loop**

<set_assign_stmt> ::= **assign_to** <element_desig> **from** <set_desig>

<set_copy_stmt> ::= **copy_to** <element_desig> **from** <set_desig>

<make_empty_stmt> ::= **make_empty** <element_desig>

<insert_stmt> ::= **insert** <element_desig> **into** <set_desig>

<remove_stmt> ::= **remove** <element_desig> **from** <set_desig>

<union_stmt> ::= <element_desig> **is_union_of** <set_desig> [, <set_desig>]*

<intersect_stmt> ::= <element_desig> **is_intersection_of** <set_desig> [, <set_desig>]*

<complement_stmt> ::= <element_desig> **is_complement_of** <set_desig$_1$> **wrt** <set_desig$_2$>

Inverse Syntax

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

Syntax of Names and Designators

<char_seg> ::= <string of letters and/or digits>

<param_seg> ::= $<ordinal_number>

<pattern_seg> ::= <char_seg> | <param_seg>

<dict_class_entry> ::= <pattern_seg> [_<pattern_seg>]*

<actual_name> ::= <char_seg> [_<char_seg>]*

<dict_instance_entry> ::= <actual_name> |
<actual_name> '[' <subscript_decl> ']'

<subscript_decl> ::= <subscript_pool_name> [, <subscript_pool_name>]*

<class_name> ::= [<scope>] <actual_name> |
CLASS | ATTRIBUTE | MAP | SET

<element_name> ::= <actual_name> | <subscripted_name>

<subscripted_name> ::= <actual_name> '[' <subscript> ']'

<subscript> ::= <subscript_value> [,<subscript_value>]*

<ADAMS_var> ::= <actual_name>

<variable_list> ::= <ADAMS_var> [, <variable_list>]

<variable_decl_stmt> ::= **ADAMS_var** <variable_list>

<element_desig> ::= [<scope>] <element_name> | <variable_name> |
<element_desig>.<map_desig> | <ADAMS_var>

<variable_name> ::= **var** <host_language_variable>

<attr_desig> ::= <element_desig>

<map_desig> ::= <element_desig>

<set_desig> ::= <element_desig> | <enumerated_set> |
NULLSET | <retrieval_set> |
<element_desig>**->**<synonym>

<range> ::= <subscript_value> .. <subscript_value>

<range_subscript> ::= <range> [, <range>]*

<enumeration_elem> ::= <element_name> |
<actual_name>'['<range_subscript>']'

<enumerated_set> ::= '{' [<enumeration_elem> [, <enumeration_elem>]*]* '}'

<var_assign_stmt> ::= <ADAMS_var> **denotes** [**var**] <element_desig>

Other Dictionary Syntax

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

<scope> ::= SYSTEM | TASK | USER | LOCAL

<rescope_stmt> ::= **rescope** <entry_type> <dict_entry> as <scope>

<erase_entry_stmt> ::= **erase** <entry_type> <dict_entry>

<entry_type> ::= CLASS | INSTANCE | CODOMAIN | SUBSCRIPT_POOL

<dict_entry> ::= <dict_class_entry> | <dict_inst_entry>

Transaction Syntax

<start_trans_stmt> ::= **tr_start** <trans_desig>

<end_trans_stmt> ::= **tr_end** <trans_desig>

<abort_stmt> ::= **abort** <trans_desig>

<lock_stmt> ::= **lock** <element_desig>

<unlock_stmt> ::= **unlock** <element_desig>

Appendix 3: YACC version of ADAMS Grammar

The following is the Grammar that was used by YACC to develop the parser for the preprocessor. It recognizes the same language as the official ADAMS BNF grammar in appendix 1, but cannot use the same grammar, because of parsing difficulties.

There are a number of differences between the formal BNF and YACC grammars. It is generally easier to follow the BNF grammar, so the major deviations in the YACC grammar will be discussed here.

(1) FOR_EACH loop_body statement: The ADAMS BNF allows for any mixture of C code or ADAMS statements within the loop_body. The YACC grammar allows only ADAMS statements there. This is because the parser never sees any C code (in legal ADAMS statements); the C code is allowable, but is just not reflected in the YACC grammar.

(2) error stmts in adams_stmt production. YACC has extra productions to handle error conditions. These are not present in the BNF grammar, since that only deals with defining allowable strings in the language. It is desirable to be able to handle error conditions gracefully in YACC, and so the need for these productions.

(3) The optional clauses in the declarations have to be handled differently here to eliminate ambiguity. The optional clauses before each clause cause problems—in order to prevent trailing commas, or multiple commas in a row, all the clauses in a set have to grouped together.

```
/*****************************************************************************/
                    Main Body
/*****************************************************************************/
adams_body:                                          /* empty string  */
               |   adams_body adams_stmt
               ;

/*****************************************************************************/
                    ADAMS statements
/*****************************************************************************/

adams_stmt:   STMT_BEGIN   abort_stmt          STMT_END
       |      STMT_BEGIN   add_codomain_method STMT_END
       |      STMT_BEGIN   attr_assign_stmt    STMT_END
       |      STMT_BEGIN   attr_decl_stmt      STMT_END
       |      STMT_BEGIN   codomain_decl_stmt  STMT_END
       |      STMT_BEGIN   close_stmt          STMT_END
       |      STMT_BEGIN   complement_stmt     STMT_END
       |      STMT_BEGIN   delete_elem_stmt    STMT_END
       |      STMT_BEGIN   end_loop_stmt       STMT_END
       |      STMT_BEGIN   extend_pool_stmt    STMT_END
       |      STMT_BEGIN   end_trans_stmt      STMT_END
       |      STMT_BEGIN   erase_entry_stmt    STMT_END
       |      STMT_BEGIN   fetch_stmt          STMT_END
       |      STMT_BEGIN   insert_stmt         STMT_END
       |      STMT_BEGIN   intersect_stmt      STMT_END
       |      STMT_BEGIN   lock_stmt           STMT_END
       |      STMT_BEGIN   looping_stmt        STMT_END
```

```
   |   STMT_BEGIN    make_empty_stmt     STMT_END
       |      STMT_BEGIN    map_decl_stmt       STMT_END
       |      STMT_BEGIN    open_stmt           STMT_END
       |      STMT_BEGIN    remove_stmt         STMT_END
       |      STMT_BEGIN    rescope_stmt        STMT_END
       |      STMT_BEGIN    set_assign_stmt           STMT_END
       |      STMT_BEGIN    set_copy_stmt       STMT_END
       |      STMT_BEGIN    start_trans_stmt    STMT_END
       |      STMT_BEGIN    store_stmt          STMT_END
       |      STMT_BEGIN    subscript_pool_decl_stmt   STMT_END
       |      STMT_BEGIN    union_stmt          STMT_END
       |      STMT_BEGIN    unlock_stmt         STMT_END
       |      STMT_BEGIN    var_assign_stmt     STMT_END
       |      STMT_BEGIN    var_decl_stmt       STMT_END
       |      STMT_BEGIN    elem_inst_stmt      STMT_END
       |      STMT_BEGIN    set_inst_stmt       STMT_END
       |      STMT_BEGIN    class_decl_stmt     STMT_END
       |      STMT_BEGIN    set_decl_stmt       STMT_END
       |      STMT_BEGIN    error  STMT_END
       |      STMT_BEGIN    error  R_PAREN
       |      STMT_BEGIN    error  SEMI
       ;

/*****************************************************************************/
                    open/close statements
/*****************************************************************************/

open_stmt:   OPEN actual_name
               ;

close_stmt:   CLOSE actual_name
               ;

/*****************************************************************************/
                    Codomains
/*****************************************************************************/

codomain_decl_stmt: dict_class_entry ISA CODOMAIN comma_opt membership_clause
                    cod_decl_stmt_options
               ;

codomain_name:   actual_name
               ;

membership_clause:   VALIDATED BY codomain_method_def
               |   CONSISTING OF CODOMAIN_VALUE
               ;

access_method_clause:   FETCH COLON codomain_method_def STORE COLON codomain_method_def
               ;

other_method_clause:   method_name COLON codomain_method_def
               ;

method_name:   actual_name
               ;

undefined_clause:   UDF EQUAL literal_value
               ;

unknown_clause:   UKN EQUAL literal_value
               ;
```

```
literal_value:   LITERAL_VALUE
               ;

codomain_method_def:   extern_def_codomain_method
               ;

extern_def_codomain_method:   EXTERNAL actual_name
               ;

cod_decl_stmt_options:   cod_decl_stmt_opt_choices
               |   cod_decl_stmt_options cod_decl_stmt_opt_non_empt
               ;

cod_decl_stmt_opt_choices:
               |   cod_decl_stmt_opt_non_empt
               ;

cod_decl_stmt_opt_non_empt:   COMMA access_method_clause
               |   COMMA other_method_clause
               |   COMMA undefined_clause
               |   COMMA unknown_clause
               |   COMMA scope_clause
               |   access_method_clause
               |   other_method_clause
               |   undefined_clause
               |   unknown_clause
               |   scope_clause
               ;

subscript_pool_decl_stmt:   dict_inst_entry_no_sub INSTANTIATES_A SUBSCRIPT_POOL OF
                    codomain_name VALUES
               ;

extend_pool_stmt:   ADD subscript_value TO actual_name POOL
               ;

add_codomain_method:   ADD METHOD TO actual_name CODOMAIN method_name COLON
                    codomain_method_def
               ;

/*****************************************************************************/
                    Attributes
/*****************************************************************************/

attr_decl_stmt:   VAR dict_class_entry ISA ATTRIBUTE comma_opt WITH IMAGE
                    codomain_image association_clause_opt_many decl_stmt_options
               ;

attr_decl_stmt:   dict_class_entry ISA ATTRIBUTE comma_opt WITH IMAGE
                    codomain_image association_clause_opt_many decl_stmt_options
               ;

codomain_image:   codomain_name
               |   param_seg
               ;

fetch_stmt:   FETCH INTO host_var FROM value_desig
               |   FETCH FROM value_desig INTO host_var
               ;

store_stmt:   STORE FROM host_expr INTO value_desig
               |   STORE INTO value_desig FROM host_expr
               ;
```

```
attr_assign_stmt:   ASSIGN INTO element_desig PERIOD attr_desig FROM value_desig
                ;

host_var:   actual_name
                ;

host_expr:   actual_name
                ;

value_desig:   element_desig PERIOD attr_desig
                |   literal_value
                ;

/*****************************************************************************/
                    Maps
/*****************************************************************************/

map_decl_stmt:   map_decl_stmt_start MAP_TOKEN comma_opt WITH IMAGE class_name
                    association_clause_opt_many decl_stmt_options
                ;

map_decl_stmt_start:   VAR dict_class_entry ISA
                |   dict_class_entry ISA
                ;

/*****************************************************************************/
                    Classes
/*****************************************************************************/

class_decl_stmt:   VAR dict_class_entry ISA super_class class_decl_body
                ;

class_decl_stmt:   dict_class_entry ISA super_class class_decl_body
                ;

class_decl_body:   FORWARD
                |   association_clause_opt_many decl_stmt_options
                ;

elem_inst_stmt_short:   elem_inst_stmt_short1 scope_clause
                |   elem_inst_stmt_short1 COMMA scope_clause
                |   elem_inst_stmt_short1
                ;

elem_inst_stmt_short1:   VAR dict_inst_entry_sub INSTANTIATES_A class_name
                |   dict_inst_entry_sub INSTANTIATES_A class_name
                |   VAR dict_inst_entry_no_sub INSTANTIATES_A class_name
                |   dict_inst_entry_no_sub INSTANTIATES_A class_name
                ;

elem_inst_stmt_long:VAR dict_inst_entry_sub INSTANTIATES_A class_name
                        AND_class_many scope_clause_opt
                |    dict_inst_entry_sub INSTANTIATES_A class_name
                        AND_class_many scope_clause_opt
                |    VAR dict_inst_entry_no_sub INSTANTIATES_A class_name
                        AND_class_many scope_clause_opt
                |   dict_inst_entry_no_sub INSTANTIATES_A class_name
                        AND_class_many scope_clause_opt
                ;

elem_inst_stmt:   elem_inst_stmt_short
                |   elem_inst_stmt_long
                ;
```

```
super_class:   super_class1
               |   super_class2
               ;

super_class1:   CLASS_TOKEN
               ;

super_class2:   class_name
               |   super_class2 AND class_name
               ;

association_clause:   HAVING synonym EQUAL association_set
               |   HAVING association_set
               ;

synonym:   actual_name
               ;

association_set:   set_desig
               |   clustered_attr_enum
               ;

clustered_attr_enum:   L_CURL L_PAREN attr_cluster R_PAREN attr_cluster_opt R_CURL
               ;

attr_cluster_opt:
               |   attr_cluster_opt COMMA attr_cluster
               ;

attr_cluster:   L_PAREN attr_cluster R_PAREN
               |   attr_cluster COMMA attr_cluster
               |   attr_desig
               ;

restrict_clause:   PROVIDED POUND predicate POUND
               ;

delete_elem_stmt:   DELETE element_desig
               ;

/*****************************************************************************/
                    Predicates
/*****************************************************************************/

predicate:   disjunct
               |   predicate OR disjunct
               ;

disjunct:   conjunct
               |   disjunct AND conjunct
               ;

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

term:   equality_comparison
               |   order_comparison
               ;

equality_comparison:   element equality_test element
               |   data_value equality_test data_value
```

```
                ;

order_comparison:   data_value order_test data_value
                ;

element:   free_var
                |   element_desig
                ;

data_value:   element PERIOD attr_desig
                |   LITERAL_VALUE
                ;

equality_test:   EQUAL
                |   NOT_EQ
                ;

order_test:   LESS_THAN
                |   LESS_EQ
                |   GREATER_THAN
                |   GREATER_EQ
                ;

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

free_var:   FREE_VAR
                ;

logical_var:   bound_var
                |   free_var
                ;

bound_var:   ADAMS_var
                ;

/*****************************************************************************/
                     Sets
/*****************************************************************************/

set_decl_stmt:   set_decl_stmt_start SET_TOKEN OF set_type ELEMENTS options
                ;

set_decl_stmt_start:   dict_class_entry ISA
                |   VAR dict_class_entry ISA
                ;

set_type:   class_name
                |   ATTRIBUTE
                |   MAP_TOKEN
                |   SET_TOKEN
                |   CLASS_TOKEN
                ;

set_inst_stmt:   elem_inst_stmt_short comma_opt initial_clause comma_opt
                    scope_clause_opt
                ;

initial_clause:   CONSISTING OF set_desig
                ;

looping_stmt:   FOR_EACH ADAMS_var IN set_desig DO loop_body
```

```
                ;

loop_body:
                |   adams_stmt
                |   loop_body adams_stmt
                ;

end_loop_stmt:   EXIT_LOOP
                ;

set_assign_stmt:   ASSIGN_TO element_desig FROM set_desig
                ;
set_copy_stmt:   COPY_TO element_desig FROM set_desig
                ;

make_empty_stmt:   MAKE_EMPTY element_desig
                ;

insert_stmt:   INSERT element_desig INTO set_desig
                ;

remove_stmt:   REMOVE element_desig FROM set_desig
                ;

union_stmt:   union_stmt_start
                |   union_stmt COMMA set_desig
                ;

union_stmt_start:   element_desig IS_UNION_OF set_desig
                ;

intersect_stmt:   intersect_stmt_start
                |   intersect_stmt COMMA set_desig
                ;

intersect_stmt_start:   element_desig IS_INTERSECTION_OF set_desig
                ;

complement_stmt:   element_desig IS_COMPLEMENT_OF set_desig WRT set_desig
                ;

/*****************************************************************************/
                    Inverses
/*****************************************************************************/

retrieval_set:   L_CURL bound_var IN set_desig BAR predicate R_CURL
                ;

/*****************************************************************************/
                    ADAMS Names and Designators
/*****************************************************************************/

char_seg:   CHARS
                ;

param_seg:   PARAM
                ;

pattern_seg:   param_seg
                |   pattern_seg UNDER param_seg
                ;

actual_name:   char_seg
```

```
                ;

subscript_decl:   subscript_pool_name
                |   subscript_decl COMMA subscript_pool_name
                ;

subscript_pool_name:   actual_name
                ;

subscript:   subscript_value
                |   subscript COMMA subscript_value
                ;

subscript_range:   range
                |   subscript COMMA range
                |   subscript_range COMMA range
                |   subscript_range COMMA subscript_value
                ;

subscript_value:   actual_name
                ;

subscripted_name:   actual_name L_SQUARE subscript R_SQUARE
                ;

subscripted_range_name:   actual_name L_SQUARE subscript_range R_SQUARE
                ;

class_name:   dict_class_entry
                |   scope dict_class_entry
                |   param_seg
                |   scope param_seg
                ;

element_name:   actual_name
                |   scope actual_name
                |   subscripted_name
                |   scope subscripted_name
                ;

ADAMS_var:   actual_name
                ;

var_list:   ADAMS_var
                |   var_list COMMA ADAMS_var
                ;

var_decl_stmt:   ADAMS_VAR var_list
                ;

element_desig:   element_name
                |   variable_name
                |   element_desig PERIOD map_desig
                ;

variable_name:   VAR actual_name
                ;

attr_desig:   element_desig
                ;

map_desig:   element_desig
                ;
```

```
set_desig:   element_desig
               |   enumerated_set
               |   retrieval_set
               |   element_name SET_ASSOC_OP synonym
               |   NULLSET
               ;

range:   subscript_value RANGE subscript_value
               ;

enumerated_set:   L_CURL enumerated_name_many R_CURL
               |   L_CURL R_CURL
               ;

enumerated_name_many:   element_name
               |   subscripted_range_name
               |   enumerated_name_many COMMA element_name
               |   enumerated_name_many COMMA subscripted_range_name
               ;

var_assign_stmt:   ADAMS_var DENOTES element_desig
               ;

scope_clause:   SCOPE IS scope
               ;

scope:   SYSTEM_TOKEN
               |   TASK_TOKEN
               |   USER_TOKEN
               |   LOCAL_TOKEN
               ;

rescope_stmt:   RESCOPE entry_type element_name AS scope
               ;

erase_entry_stmt:   ERASE entry_type dict_entry
               ;

entry_type:   CLASS_TOKEN
               |   INSTANCE
               |   CODOMAIN
               |   SUBSCRIPT_POOL
               ;

/*****************************************************************************/
                    Dictionary
/*****************************************************************************/

dict_entry:   dict_class_entry_param
               |   dict_inst_entry_sub
               |   actual_name
               |   scope actual_name
               ;

dict_class_entry:   dict_class_entry_actual
               |   dict_class_entry_param
               ;

dict_class_entry_actual:   actual_name
               ;

dict_class_entry_param:   pattern_seg UNDER_CHARS
               |   pattern_seg UNDER_CHARS_UNDER pattern_seg
```

```
                |   dict_class_entry_param UNDER_CHARS_UNDER pattern_seg
                |   dict_class_entry_param UNDER_CHARS
                |   dict_class_entry_param UNDER pattern_seg
                ;

dict_inst_entry:   dict_inst_entry_no_sub
                |   dict_inst_entry_sub
                ;

dict_inst_entry_no_sub:   actual_name
                |   dict_inst_entry_semis
                ;

dict_inst_entry_sub:   subscripted_name
                ;

dict_inst_entry_semis:   SEMI_CHARS
                |   actual_name SEMI_CHARS
                |   dict_inst_entry_semis CHARS_UNDER
                |   dict_inst_entry_semis SEMI_CHARS
                ;

/*****************************************************************************/
                    Transactions
/*****************************************************************************/

start_trans_stmt:   TR_START CHARS
                ;

end_trans_stmt:   TR_END CHARS
                ;

abort_stmt:   ABORT CHARS
                ;

lock_stmt:   LOCK element_desig
                ;

unlock_stmt:   UNLOCK element_desig
                ;

/*****************************************************************************/
                    Declaration statement options
/*****************************************************************************/

options:   association_clause_opt_many decl_stmt_options
                ;

decl_stmt_options:                                          /* empty string  */
                |   decl_stmt_opt_non_empt
                |   COMMA decl_stmt_opt_non_empt
                |   decl_stmt_options comma_opt decl_stmt_opt_non_empt
                ;

decl_stmt_opt_non_empt:   scope_clause
                |   restrict_clause
                ;

/*****************************************************************************/
                    Optional clauses
/*****************************************************************************/
scope_clause_opt:                                               /* empty string  */
                |   scope_clause
```

```
                |   COMMA scope_clause
                ;

association_clause_opt_many:                                        /* empty string  */
                |   association_clause
                |   association_clause_opt_many association_clause
                |   association_clause_opt_many COMMA association_clause
                ;

AND_class_many:   AND class_name
                |   AND_class_many AND class_name
                ;

comma_opt:                                              /* empty string  */
                |   COMMA
                ;
```

Appendix 4: LEX Tokens

The following is a list of tokens that are used in the LEX program; they are included here since they are used in appendix 2, in the YACC grammar.

This first list of tokens are those that are formed from the literal characters in the name of the token, with either upper or lower case allowed in each position (ie, ALL could be formed by "ALL", "ALl", "AlL", "All", "aLL", "aLl", "alL", all"). Several (CLASS_TOKEN for example) are of this pattern, but not for the exact token name ("_TOKEN" had to be added to avoid conflicts with other names in the preprocessor). The words in the ADAMS language that tokenizes into them is listed in parenthesis following these few exceptions.

```
ABORT, ACCESS_NAME, ADAMS_VAR, ADD, ALL, AND, AS, ASSIGN, ASSIGN_TO,
ATTRIBUTE, BAR, BELONGS, BY, CLASS_TOKEN (CLASS), CLOSE, CODOMAIN,
CODOMAIN_VALUE, CONSISTING, COPY_TO, DELETE, DENOTES, DENOTES_A, DO,
ELEMENTS, EQUAL, ERASE, EXIT_LOOP, EXISTS, EXTERNAL, FETCH, FOR_EACH,
FORWARD, FREE_VAR, FROM, HAVING, IMAGE, INSERT, INSTANCE,
INSTANTIATES_A, IN, INTO, IS, ISA, IS_COMPLEMENT_OF, IS_INTERSECTION_OF,
IS_UNION_OF, LOCK, LOCAL_TOKEN (LOCAL), MAP_TOKEN (MAP), MAKE_EMPTY,
METHOD, NAME_VAR, NOT_EQ, NULLSET, OF, OR, OPEN, PLUS, POOL, PROVIDED,
REMOVE, RESCOPE, SCOPE, SET_TOKEN (SET), SET_ASSOC_OP, STORE, STR_CONST,
SUBSCRIPT_POOL, SYSTEM_TOKEN (SYSTEM), TASK_TOKEN (TASK), TO, TR_START,
TR_END, UDF, UKN, UNDEFINED, UNLOCK, USE, USER_TOKEN (USER), USING,
VALIDATED, VALUE, VALUES, VAR, VIEW, WITH, WHICH, WRT.
```

The standard UNIX notation for regular expressions is used to describe the strings that don't match the default pattern for this list.

```
CHARS       [a-zA-Z]+
CHARS_UNDER      [a-zA-Z]+"_"
COLON       ":"
COMMA       ","
GREATER_EQ ">="
GREATER_THAN     ">"
L_CURL           "{"
LESS_EQ          "<="
LESS_THAN   "<"
L_PAREN          "("
L_SQUARE    "["
PARAM       "$"[1-9][0-9]*
PERIOD           "."
POUND       "#"
QUOTE       """
```

```
R_CURL           "}"
R_PAREN          ")"
R_SQUARE   "]"
RANGE      ".."
SEMI       ":"
SEMI_CHARS ":"[a-zA-Z]+
SING_QUOTE "’"
STMT_BEGIN "<<"
STMT_END   ">>"
UNDER      "_"
UNDER_CHARS     "_"[a-zA-Z]+
UNDER_CHARS_UNDER     "_"[a-zA-Z]+
```

Appendix 5: Sample ADAMS program and C translation.

In this appendix, we present a more extensive ADAMS program, along with the C program that is generated when this source file is input to the preprocessor. First the ADAMS/C source file:

```
                                    /*  sample ADAMS/C code  */

#include <stdio.h>

main ()
{
int   i;
char  xx [30];                      /* defs for adams_vars  */
char  f_name [30], l_name [30], addr [50], phone [10];

<< adams_var x, y >>
<< open_adams 3  >>

printf ("about to declare c class\n");
<< c isa CLASS, having c_attr =
           {c_f_name, c_l_name, c_addr, c_phone}, SCOPE is USER >>
printf ("about to instantiate c_inst \n");
<< c_inst instantiates_a c >>

printf ("about to decl c_set \n");
<< c_set isa SET of c ELEMENTS >>
printf ("about to instantiate c_set_inst \n");
<< c_set_inst1 instantiates_a c_set >>
<< c_set_inst2 instantiates_a c_set >>
<< c_set_inter instantiates_a c_set >>
<< c_set_union instantiates_a c_set >>

for (i = 0; i < 4; i++)
      {
      printf ("about to scanf for var name\n");
      scanf ("%s", xx);
      << var xx instantiates_a c >>
      << insert var xx into c_set_inst1 >>
      scanf ("%s", f_name);
      scanf ("%s", l_name);
      scanf ("%s", addr);
      scanf ("%s", phone);
      << store from f_name into var    xx.c_f_name >>
      << store from l_name into var    xx.c_l_name >>
      << store from phone into var     xx.c_phone >>
      << store from addr into varxx.c_addr >>
```

```
   << insert var xx into c_set_inst2 >>
      if (strcmp ("John", f_name) )
           << insert var   xx into c_set_inst1 >>
      }

printf ("\nelements of set 1 are:\n");
<<  FOR_EACH y in c_set_inst1 DO
      << fetch into f_name from y.c_f_name >>
      << fetch into l_name from y.c_l_name >>
      << fetch into addr from y.c_addr >>
      << fetch into phone from y.c_phone >>
>>

printf ("\nelements of set 2 are:\n");
<<  FOR_EACH y in c_set_inst2 DO
      << fetch into f_name from y.c_f_name >>
      << fetch into l_name from y.c_l_name >>
      << fetch into addr from y.c_addr >>
      << fetch into phone from y.c_phone >>
>>

<< c_set_inter is_intersection_of c_set_inst1, c_set_inst2 >>
printf ("\nelements of intersection set are:\n");
<<  FOR_EACH y in c_set_inter DO
      << fetch into f_name from y.c_f_name >>
      << fetch into l_name from y.c_l_name >>
      << fetch into addr from y.c_addr >>
      << fetch into phone from y.c_phone >>
>>

<< c_set_union is_union_of c_set_inst1, c_set_inst2 >>
printf ("\nelements of union set are:\n");
<<  FOR_EACH y in c_set_union DO
      << fetch into f_name from y.c_f_name >>
      << fetch into l_name from y.c_l_name >>
      << fetch into addr from y.c_addr >>
      << fetch into phone from y.c_phone >>
>>

<< close_adams 3 >>
}
```

Next we have the C translation of the ADAMS/C source code:

```
#define    _A_MAX_ARGS_PER_STMT  20
#define    _A_FETCH_BUFF_SZ 50
#define    _A_MAX_NUM_SUBS  20
#include   "uid_cc.h"
#define    _A_IDLEN   8
#include   "dict_cc.h"
```

```
extern     void  _A_read_symbol_table (), _A_read_num_entries ();
extern     void  _A_attach_dict (), _A_release_dict ();
extern     int   _A_check_symbol_table ();
extern     void  _A_inter_LOCK (), _A_inter_UNLOCK ();
extern     int   _A_check ();
extern     int   _A_run_time_lock (), _A_run_time_unlock ();
extern     char  *malloc (), *strcpy ();
int   _ADAMS_STATUS;
int   _A_c_const_uid ();
#include   "symbol.h"
#include   "indexglue.h"
#include   "indexman.h"
_A_uid_string    _A_uid_name, _A_set_uid, _A_fn_uid, _A_class_uid;
_A_uid_string    _A_inst_uid;
_A_uid_string    _A_var_name, _A_map_elem, _A_map_fn, _A_map_target;
_A_uid_string    _A_temp_uid0, _A_temp_uid1, _A_temp_uid2, _A_temp_uid3;
_A_uid_string    _A_subs_uid, _A_subs_temp;
_A_D_LEVEL _A_d_level = (_A_D_LEVEL) 0;
char  _A_args [_A_MAX_ARGS_PER_STMT] [_A_IDLEN + 1];
int   uid = 853, tid = 123;
int   _A_int_index;
double     _A_unique_double;
int   _A_num_subs, _A_subs_int [_A_MAX_NUM_SUBS];
_A_ENTRYDEF      _A_def, _A_temp_def, _A_set_def;
_A_ENTRYTYPE     _A_entry_type;
char  **_A_var_ptr;
char  _A_foo [100];
char  _A_foo2 [100];


                                /*  sample ADAMS/C code  */

#include <stdio.h>

main ()
{
int   i;
char  xx [30];              /* defs for adams_vars  */
char  f_name [30], l_name [30], addr [50], phone [10];

_A_uid_string    _A_uid_x;
_A_uid_string    _A_uid_y;

          /*  << adams_var x , y >>   */
/*-------------------------------------------------------------------*/
_ADAMS_STATUS = 1;
_main();
_A_attach_dict (tid, uid, "new_ex",
          "/at0/pkb4h/yacc/adams/run_tests/adamsdict", 0);
_A_sym_tbl_open ("new_ex_sym" );
/*_A_c_check_symbol_table ();*/
```

```
_A_def.type = _A_NDEF;
_A_temp_def.type = _A_NDEF;
_A_set_def.type = _A_NDEF;

          /*  << open_adams 3 >>   */
/*-----------------------------------------------------------------*/

printf ("about to declare c class\n");
_ADAMS_STATUS = 1;
          /*   declare class c of type CLASS    */
_A_set_deftype (&_A_def, _A_CLASSDEF);
_A_add_superclass (&_A_def, "CLASS", _A_USER);
_A_c_const_uid (_A_class_uid, _A_MAPSET);
strcpy (_A_set_uid, "0");
_A_uid_instant (_A_set_uid, _A_class_uid);
                /*   insert MAP elements into set     */
_A_d_level = _A_USER;
if (!_A_c_check ("c_f_name", _A_uid_name, _A_INST, &_A_d_level))
     {
     printf ("c_check failed for c_f_name in class_decl:\n");
     _ADAMS_STATUS = 0;
     }
_A_set_insert ( _A_uid_name, _A_set_uid);
                /*   insert ATTR elements into set    */
_A_d_level = _A_USER;
if (!_A_c_check ("c_l_name", _A_uid_name, _A_INST, &_A_d_level))
     {
     printf ("c_check failed for c_l_name in class_decl:\n");
     _ADAMS_STATUS = 0;
     }
_A_set_insert ( _A_uid_name, _A_set_uid);
                /*   insert ATTR elements into set    */
_A_d_level = _A_USER;
if (!_A_c_check ("c_addr", _A_uid_name, _A_INST, &_A_d_level))
     {
     printf ("c_check failed for c_addr in class_decl:\n");
     _ADAMS_STATUS = 0;
     }
_A_set_insert ( _A_uid_name, _A_set_uid);
                /*   insert ATTR elements into set    */
_A_d_level = _A_USER;
if (!_A_c_check ("c_phone", _A_uid_name, _A_INST, &_A_d_level))
     {
     printf ("c_check failed for c_phone in class_decl:\n");
     _ADAMS_STATUS = 0;
     }
_A_set_insert ( _A_uid_name, _A_set_uid);
                /*   add ATTR set to definition */
_A_add_attrset (&_A_def, NULL, _A_set_uid, _A_USER, "c_attr");
strcpy (_A_uid_name, "1");
                /*   add class definition to dict now:     */
```

```
if (!_A_add_entry ("c", _A_uid_name, &_A_def, _A_CLASS, _A_USER))
{
printf ("_A_add_entry failed for class c of type  \n");
     _ADAMS_STATUS = 0;
     }

     /*  << c isa CLASS , having c_attr =
          { c_f_name , c_l_name , c_addr , c_phone } , SCOPE is USER >> */
/*-------------------------------------------------------------------*/
printf ("about to instantiate c_inst \n");
_ADAMS_STATUS = 1;
          /*   declare element c_inst of class c*/
strcpy (_A_inst_uid, "2");
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "c", _A_USER);
               /*handle argument # 1 */
               /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (0) );
if (!_A_add_entry ("c_inst", _A_inst_uid, &_A_def, _A_INST, _A_USER))
     {
     printf ("_A_add_entry failed for c_inst of class c\n");
     _ADAMS_STATUS = 0;
     }
_A_uid_instant (_A_inst_uid, _A_args [1]);

          /*  << c_inst instantiates_a c >>   */
/*-------------------------------------------------------------------*/

printf ("about to decl c_set \n");
_ADAMS_STATUS = 1;
          /*   declare set c_set of c elements  */
strcpy (_A_uid_name, "3");
_A_set_deftype (&_A_def, _A_CLASSDEF);
_A_add_superclass (&_A_def, "SET", _A_USER);
_A_add_elementclass (&_A_def, "c", _A_USER);
if (!_A_add_entry ("c_set", _A_uid_name, &_A_def, _A_CLASS, _A_USER))
     {
     printf ("_A_add_entry failed for set c_set of c elements\n");
     _ADAMS_STATUS = 0;
     }

          /*  << c_set isa SET of c ELEMENTS >>   */
/*-------------------------------------------------------------------*/
printf ("about to instantiate c_set_inst \n");
_ADAMS_STATUS = 1;
          /*   declare element c_set_inst1 of class c_set*/
strcpy (_A_inst_uid, "4");
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "c_set", _A_USER);
               /*handle argument # 1 */
               /* name was found at parse time */
```

```
strcpy (_A_args [1], _A_sym_tbl_ref (1) );
if (!_A_add_entry ("c_set_inst1", _A_inst_uid, &_A_def, _A_INST, _A_USER))
     {
     printf ("_A_add_entry failed for c_set_inst1 of class c_set\n");
     _ADAMS_STATUS = 0;
     }
_A_uid_instant (_A_inst_uid, _A_args [1]);

          /*  << c_set_inst1 instantiates_a c_set >>   */
/*-------------------------------------------------------------------*/
_ADAMS_STATUS = 1;
          /*   declare element c_set_inst2 of class c_set*/
strcpy (_A_inst_uid, "5");
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "c_set", _A_USER);
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (1) );
if (!_A_add_entry ("c_set_inst2", _A_inst_uid, &_A_def, _A_INST, _A_USER))
     {
     printf ("_A_add_entry failed for c_set_inst2 of class c_set\n");
     _ADAMS_STATUS = 0;
     }
_A_uid_instant (_A_inst_uid, _A_args [1]);

          /*  << c_set_inst2 instantiates_a c_set >>   */
/*-------------------------------------------------------------------*/
_ADAMS_STATUS = 1;
          /*   declare element c_set_inter of class c_set*/
strcpy (_A_inst_uid, "6");
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "c_set", _A_USER);
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (1) );
if (!_A_add_entry ("c_set_inter", _A_inst_uid, &_A_def, _A_INST, _A_USER))
     {
     printf ("_A_add_entry failed for c_set_inter of class c_set\n");
     _ADAMS_STATUS = 0;
     }
_A_uid_instant (_A_inst_uid, _A_args [1]);

          /*  << c_set_inter instantiates_a c_set >>   */
/*-------------------------------------------------------------------*/
_ADAMS_STATUS = 1;
          /*   declare element c_set_union of class c_set*/
strcpy (_A_inst_uid, "7");
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "c_set", _A_USER);
                /*handle argument # 1 */
                /* name was found at parse time */
```

```
strcpy (_A_args [1], _A_sym_tbl_ref (1) );
if (!_A_add_entry ("c_set_union", _A_inst_uid, &_A_def, _A_INST, _A_USER))
     {
     printf ("_A_add_entry failed for c_set_union of class c_set\n");
     _ADAMS_STATUS = 0;
     }
_A_uid_instant (_A_inst_uid, _A_args [1]);

          /*  << c_set_union instantiates_a c_set >>   */
/*-----------------------------------------------------------------------*/

for (i = 0; i < 4; i++)
     {
     printf ("about to scanf for var name\n");
     scanf ("%s", xx);
     _ADAMS_STATUS = 1;
          /*   declare element VAR xx of class c*/
_A_uid_getuid (_A_inst_uid);
_A_set_deftype (&_A_def, _A_INSTDEF);
_A_add_instclass (&_A_def, "c", _A_USER);
               /*handle argument # 1 */
               /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (0) );
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if (!_A_add_entry (_A_var_name, _A_inst_uid, &_A_def, _A_INST, _A_d_level))
     {
     printf ("_A_add_entry failed for adams_var %s of class c\n", xx);
     _ADAMS_STATUS = 0;
     }
_A_uid_instant (_A_inst_uid, _A_args [1]);

          /*  << var xx instantiates_a c >>   */
/*-----------------------------------------------------------------------*/
     _ADAMS_STATUS = 1;
               /*handle argument # 0 */
               /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_args [0], _A_uid_name);
               /*handle argument # 1 */
               /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (2) );
_A_set_insert (_A_args [0], _A_args [1]);
          /*  << insert var xx into c_set_inst1 >>   */
```

```
/*-----------------------------------------------------------------*/
     scanf ("%s", f_name);
     scanf ("%s", l_name);
     scanf ("%s", addr);
     scanf ("%s", phone);
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
               /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_temp_uid1, _A_uid_name);
                                /*handle temp arg 0*/
               /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (3) );
_A_attr_insert (_A_temp_uid0, _A_temp_uid1, f_name);

          /*  << store from f_name into var xx . c_f_name >>   */
/*-----------------------------------------------------------------*/
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
               /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_temp_uid1, _A_uid_name);
                                /*handle temp arg 0*/
               /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (4) );
```

```
_A_attr_insert (_A_temp_uid0, _A_temp_uid1, l_name);

          /*  << store from l_name into var xx . c_l_name >>   */
/*-------------------------------------------------------------------*/
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                  /*handle temp arg 1*/
                /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_temp_uid1, _A_uid_name);
                                  /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (5) );
_A_attr_insert (_A_temp_uid0, _A_temp_uid1, phone);

          /*  << store from phone into var xx . c_phone >>   */
/*-------------------------------------------------------------------*/
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                  /*handle temp arg 1*/
                /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_temp_uid1, _A_uid_name);
                                  /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (6) );
_A_attr_insert (_A_temp_uid0, _A_temp_uid1, addr);
```

```
          /*  << store from addr into var xx . c_addr >>   */
/*-----------------------------------------------------------------------*/
     _ADAMS_STATUS = 1;
                /*handle argument # 0 */
                /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_args [0], _A_uid_name);
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (7) );
_A_set_insert (_A_args [0], _A_args [1]);
          /*  << insert var xx into c_set_inst2 >>   */
/*-----------------------------------------------------------------------*/
     if (strcmp ("John", f_name) )
          _ADAMS_STATUS = 1;
                /*handle argument # 0 */
                /* name is a var variable; */
strcpy (_A_uid_name, "");
_A_d_level = _A_LOCAL;
strcpy (_A_var_name, _A_var_name_handle (xx, &_A_d_level));
if ( _A_c_check (_A_var_name, _A_uid_name, _A_INST,&_A_d_level) == 0)
     {
     printf ("_A_c_check failed: %s not found\n", xx);
     _ADAMS_STATUS = 0;
     }
else
     strcpy (_A_args [0], _A_uid_name);
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (2) );
_A_set_insert (_A_args [0], _A_args [1]);
          /*  << insert var xx into c_set_inst1 >>   */
/*-----------------------------------------------------------------------*/
     }

printf ("\nelements of set 1 are:\n");
_ADAMS_STATUS = 1;
     /* get uid for c_set_inst1, the set of elements being looped over */
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (2) );
{
_A_uid_string   _A_loop_uid1;
```

```
strcpy (_A_loop_uid1, _A_args [1]);
if (_A_set_first_element (_A_uid_y, _A_loop_uid1 ))
                          /* FOR_loop 1 */
     do   {
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                               /*handle temp arg 1*/
                          /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                               /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (3) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, f_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                               /*handle temp arg 1*/
                          /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                               /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (4) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, l_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                               /*handle temp arg 1*/
                          /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                               /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (6) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, addr, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
```

```
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                             /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                  /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (5) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, phone, _A_FETCH_BUFF_SZ);


}  while (_A_set_next_element (_A_uid_y, _A_loop_uid1));
                             /* fOR_EACH loop 1 */
_l1:;
}

          /*  << FOR_EACH y in c_set_inst1 DO
          << fetch into f_name from y . c_f_name >>
          << fetch into l_name from y . c_l_name >>
          << fetch into addr from y . c_addr >>
          << fetch into phone from y . c_phone >> >>   */
/*-----------------------------------------------------------------------*/

printf ("\nelements of set 2 are:\n");
_ADAMS_STATUS = 1;
     /* get uid for c_set_inst2, the set of elements being looped over */
                  /*handle argument # 1 */
                  /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (7) );
{
_A_uid_string   _A_loop_uid2;
strcpy (_A_loop_uid2, _A_args [1]);
if (_A_set_first_element (_A_uid_y, _A_loop_uid2 ))
                             /* FOR_loop 2 */
     do   {
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                             /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                  /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (3) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, f_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
```

```
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                 /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (4) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, l_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                 /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (6) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, addr, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                 /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (5) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, phone, _A_FETCH_BUFF_SZ);


}  while (_A_set_next_element (_A_uid_y, _A_loop_uid2));
                            /* fOR_EACH loop 2 */
_l2:;
}

          /*  << FOR_EACH y in c_set_inst2 DO
          << fetch into f_name from y . c_f_name >>
          << fetch into l_name from y . c_l_name >>
          << fetch into addr from y . c_addr >>
          << fetch into phone from y . c_phone >> >>   */
/*-----------------------------------------------------------------*/
```

```
_ADAMS_STATUS = 1;
                /*handle argument # 0 */
                /* name was found at parse time */
strcpy (_A_args [0], _A_sym_tbl_ref (8) );
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (2) );
                /*handle argument # 2 */
                /* name was found at parse time */
strcpy (_A_args [2], _A_sym_tbl_ref (7) );
                /* variable # of args--null element ends list */
strcpy (_A_args [3], "");
_A_var_ptr = &_A_args [1];
_A_set_intersect ( _A_args [0], _A_var_ptr );

     /*  << c_set_inter is_intersection_of c_set_inst1 , c_set_inst2 >>  */
/*-----------------------------------------------------------------*/
printf ("\nelements of intersection set are:\n");
_ADAMS_STATUS = 1;
     /* get uid for c_set_inter, the set of elements being looped over */
                /*handle argument # 1 */
                /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (8) );
{
_A_uid_string   _A_loop_uid3;
strcpy (_A_loop_uid3, _A_args [1]);
if (_A_set_first_element (_A_uid_y, _A_loop_uid3 ))
                            /* FOR_loop 3 */
     do   {
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                  /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                  /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (3) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, f_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                  /*handle temp arg 1*/
                            /*handle ADAMS_var */
```

```
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (4) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, l_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (6) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, addr, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (5) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, phone, _A_FETCH_BUFF_SZ);


}  while (_A_set_next_element (_A_uid_y, _A_loop_uid3));
                            /* fOR_EACH loop 3 */
_l3:;
}

          /*  << FOR_EACH y in c_set_inter DO
          << fetch into f_name from y . c_f_name >>
          << fetch into l_name from y . c_l_name >>
          << fetch into addr from y . c_addr >>
          << fetch into phone from y . c_phone >> >>   */
/*-----------------------------------------------------------------*/

_ADAMS_STATUS = 1;
                /*handle argument # 0 */
                /* name was found at parse time */
strcpy (_A_args [0], _A_sym_tbl_ref (9) );
```

```
          /*handle argument # 1 */
               /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (2) );
               /*handle argument # 2 */
               /* name was found at parse time */
strcpy (_A_args [2], _A_sym_tbl_ref (7) );
               /* variable # of args--null element ends list */
strcpy (_A_args [3], "");
_A_var_ptr = &_A_args [1];
_A_set_union ( _A_args [0], _A_var_ptr );

     /*  << c_set_union is_union_of c_set_inst1 , c_set_inst2 >>   */
/*-------------------------------------------------------------------*/
printf ("\nelements of union set are:\n");
_ADAMS_STATUS = 1;
     /* get uid for c_set_union, the set of elements being looped over */
               /*handle argument # 1 */
               /* name was found at parse time */
strcpy (_A_args [1], _A_sym_tbl_ref (9) );
{
_A_uid_string   _A_loop_uid4;
strcpy (_A_loop_uid4, _A_args [1]);
if (_A_set_first_element (_A_uid_y, _A_loop_uid4 ))
                         /* FOR_loop 4 */
     do   {
     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                              /*handle temp arg 1*/
                         /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                              /*handle temp arg 0*/
               /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (3) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, f_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                              /*handle temp arg 1*/
                         /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                              /*handle temp arg 0*/
               /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (4) );
```

```
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, l_name, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (6) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, addr, _A_FETCH_BUFF_SZ);

     _ADAMS_STATUS = 1;
strcpy (_A_uid_name, "");
strcpy (_A_fn_uid, "");
strcpy (_A_map_elem , "");
strcpy (_A_map_fn , "");
strcpy (_A_map_target , "");
                                /*handle temp arg 1*/
                            /*handle ADAMS_var */
strcpy (_A_temp_uid1, _A_uid_y);
                                /*handle temp arg 0*/
                /* name was found at parse time */
strcpy (_A_temp_uid0, _A_sym_tbl_ref (5) );
_A_attr_get_val (_A_temp_uid0, _A_temp_uid1, phone, _A_FETCH_BUFF_SZ);


}  while (_A_set_next_element (_A_uid_y, _A_loop_uid4));
                            /* fOR_EACH loop 4 */
_l4:;
}

          /*  << FOR_EACH y in c_set_union DO
          << fetch into f_name from y . c_f_name >>
          << fetch into l_name from y . c_l_name >>
          << fetch into addr from y . c_addr >>
          << fetch into phone from y . c_phone >> >>   */
/*-----------------------------------------------------------------*/

_ADAMS_STATUS = 1;
_A_set_deftype (&_A_def, _A_NDEF);
_A_close_indexes ();
_A_release_dict ();

          /*  << close_adams 3 >>   */
/*-----------------------------------------------------------------*/
}
```

Table of Contents