To: Distribution

From: J. A. Weeldreyer

Date: February 23, 1977

Subject: Logical Inquiry and Update System (LINUS)

Introduction

There is a requirement for an "End User Facility" to provide access to Multics data bases, on an ad hoc basis, for users who may not be computer specialists. Such a facility is not currently provided by the Multics Data Base Manager (MDBM), since this subsystem was designed for the application programmer. This MTB proposes a new subsystem LINUS which, together with the Report Generator Language (RGL) described in MTB-321, will fulfill this requirement.

Please forward comments or suggestions to Weeldreyer.Multics on System M, or call (602) 249-7244 or HVN 341-7244.

Overview

LINUS is designed to provide a data base query and update capability for users who may not be computer specialists, but who are willing to learn to converse with a computer in a fairly structured manner via a high level, non-procedural language. This facility is designed primarily for interactive use on an ad hoc basis. However, an extensive macro capability is provided to allow a "canned" series of operations to be performed as required.

LINUS operates as a subsystem, much like debug or a text editor, and provides the capability to retrieve and update data in a Multics Relational Data Store (MRDS) data base. (If required, LINUS may be extended, at some point in the future, to access Multics Integrated Data Store (MIDS) data bases as well.) Data to be selected is specified via expressions in the LINUS Language (LILA) which is a dialect of SEQUEL [1,2].

The LILA expressions will be translated directly into MRDS selection expressions, except for the few cases where a one-to-one mapping is not possible. Such cases are those in which a set function (e.g. max, min) is applied to the results of an inner select-from-where block. An example of such an expression is:

```
select name
from emp
where sal > max {select sal
                 from emp
                 where dept = "Shoe"}
```

Here, the LILA expression would be translated into two MRDS selection expressions: one to retrieve salary values so that a maximum can be determined, and a second to select the desired names. All LINUS data base operations will be accomplished via calls to the MRDS data sublanguage entries.

It should be noted that several features normally provided in "End User Facilities" such as the Management Data Query System (MDQS) on GCOS, will not be provided directly within LINUS. Instead, the capability is provided to place selected data into segments or multi-segment files within the Multics storage system. This allows other Multics facilities to be invoked to perform such functions as report formatting (including grouping, totaling, sorting), statistical analyses, editing, etc. There is also no data base definition capability within LINUS, since the data base and data submodel creation commands included in MRDS are very simple and easy to use.

Modifications to the MRDS

Several extensions to the MRDS selection expression will be required to support LINUS. These proposed changes are compatible with the current selection expression, and will require no changes to existing MRDS application programs.

One change is to define the -dup keyword which may be included in the -select clause for the dsl_$retrieve entry. This keyword specifies that duplicate selected values are to be returned to the caller. If omitted, duplicates will be eliminated (which is the current mode of operation). For example:

```
-select -dup x.salary
```

will return all salary values, including duplicates, while

```
-select x.salary
```

will return only unique values of salary. The -dup keyword will simplify the retrieval of data to be used for the calculation of averages, sums, etc.

Another MRDS extension will allow the use of arithmetic expressions within the -where clause. This will allow -where clauses such as:

```
-where (x.salary + x.commission) > 10000
```

A third modification will allow scalar functions to be used within the -where clause. There will be a set of pre-defined functions available, and the capability for user-defined functions will also be provided. User-defined functions will be declared via a new entry, dsl_$declare_fn. An example of this new capability is:

```
-where (index (x.addr "Arizona")) ^= 0
```

The above -where clause would return all tuples in which the address attribute contains the string "Arizona".

Three new keywords will be added: -inter, -union, and -differ. These correspond to the set operations intersection, union, and difference, respectively. They may be placed between range-select-where blocks within a selection expression. This feature allows set operations to be performed on sets of selected data. For example:

```
-range (x Phoenix_phone_book)
-select x
-union
-range (x Tucson_phone_book)
-select x
```

selects the union of the two phone books. With the addition of this feature, the MRDS becomes relationally complete.

In the original specification of the MRDS, provisions were made for universal quantifiers, and for complex ranges, i.e. the capability to have a single tuple variable range over unions, intersections, and differences of several relations. However, these features were not implemented in the initial release of MRDS. Since then, experience has shown that the class of queries in which universal quantification would be useful is extremely small. Hence, universal quantification will not be implemented in MRDS. In fact, explicit user specification of existential quantification will no longer be required. Also, complex ranges will not be implemented. The capability to specify set operations on selected data is equivalent in power, and is simpler for users to comprehend.

User Documentation

The following comprises the user documentation for LINUS.

Name: linus

This command invokes the Logical Inquiry and Update System (LINUS) which is a powerful, yet easy to use, facility for accessing a Multics Relational Data Store (MRDS) data base. LINUS provides a complete data base management capability including both retrieval and update operations. Data to be selected is specified via the LINUS Language (LILA) which is a high level, non-procedural language capable of being understood and used by individuals who are not necessarily computer specialists.

Usage

```
linus {macro_path} {optional_args}
```

where:

1. macro_path
   is an optional argument and, if present, specifies the pathname of an ASCII segment from which LINUS is to take its initial instructions. Such a set of instructions is referred to as a macro. If macro_path does not have a suffix of linus, then one is assumed. However, the suffix linus must be the last component of the name of the segment.

2. optional_args
   are character strings to be substituted for special strings in the macro segment. An optional argument may be specified only if a macro_path is also given.

If macro_path is provided, LINUS executes the requests contained in the specified segment and then waits for the user to type further requests. If macro_path is omitted, LINUS waits for the user to type a request. A discussion of LINUS macros is provided in the section, Macro Facilities.

Notes

While most users interact with LINUS through the terminal, this facility is designed to accept input through the user_input I/O switch and to transmit output through the user_output I/O switch. These switches can be controled, via the io_call command, to interface with other devices/files in addition to the user's terminal. For convenience, the LINUS description assumes that the user's input/output device is a terminal.

LINUS recognizes and handles the program_interrupt condition. Thus, the user may interrupt any request and resume the LINUS session by using the Multics command, program_interrupt. After the program_interrupt command, LINUS waits for the user to type further requests.

Selection Language

Several of the LINUS requests (e.g. modify, delete, print) operate on well defined subsets of a data base. These data base subsets are selected via expressions in the LINUS Language (LILA). LILA is a dialect of SEQUEL [1,2] and is designed for use by individuals who may not be computer programmers. The user views the data base as a set of tables containing rows and columns of data. LILA allows the selection algorithm to be specified as a series of table look-up operations, very similar to the way an individual would manually scan a set of tables for information. For example, one can envision a telephone directory as being a table with three columns of information: name, address, and phone number. This table contains one row of information for each individual listed in the directory. Normally, if one wishes to find the phone number for John C. Smith, one scans the name column for the name "Smith, John C.", and then takes the value from the phone number column in the same row. In LILA, this operation is described as:

```
select number
from phone_book
where name = "Smith, John C."
```

Various features of LILA will now be introduced via examples referencing a data base consisting of five tables which describe the operation of a department store:

emp (<u>name</u>, emp_no, dept, mgr, sal, comm)
sales (<u>dept</u>, <u>item</u>, vol)
supply (<u>supplier</u>, <u>item</u>, vol)
loc (<u>dept</u>, floor)
class (<u>item</u>, type)

The emp table contains a row for every employee, giving his name, employee number, department, manager number, salary, and commission for the last year. The sales table gives the volume of sales for every item within each department. The supply table provides the volume of each item supplied by every supplier. The loc table gives the floor on which every department is located, and the class table specifies the type of each item.

In each of the above tables, the underscores denote key columns. Every row in a table is uniquely identified by its values in the key columns. The LINUS user need not be concerned with the key column concept except when using the modify and define_temp_table requests discussed later.

The basic component of LILA is the select-from-where block, which is used to <u>select</u> column values <u>from</u> one or more tables <u>where</u> rows of the tables satisfy certain conditions. It should be noted that the indentation of the following examples is for readability only, and is not required in actual usage. In fact, the entire LILA expression may be contained in one line.

The select clause and the from clause must always be specified in a select-from-where block. The where clause of a block may be omitted, in which case all rows are returned.

Example 1: List all departments from the emp table.

```
select dept
from emp
```

A select clause may contain one or more column names, as in the previous example; or may contain a *, indicating that all columns from qualifying rows are to be selected.

Example 2: List all information pertaining to every employee whose salary is greater than $8,000.

```
select *
from emp
where sal > 8000
```

More complex conditions may be specified in the where clause, as shown in the following examples. Specifically, a where clause may contain one or more terms. Each term consists of a column name or an arithmetic expression; followed by a relational operator; followed by a column name, arithmetic expression, or constant. Allowable relational operators are:

>	greater than,
<	less than,
<=	less than or equal to (or not greater than),
>=	greater than or equal to (or not less than),
=	equal to,
^=	not equal to.

Terms within the where clause must be separated by logical operators, and may be grouped using parentheses to explicitly specify order of evaluation. Allowable logical operators are:

&	logical conjunction ("and"),
|	logical inclusive "or",
^	logical negation ("not").

Character string constants within terms must be enclosed within quotes. If a quote is to appear within a character string, a double quote must be specified.

Example 3: Find the names and salaries of employees in the toy department who work for Anderson, whose employee number is 1423.

```
          select name sal
          from emp
          where dept = "Toy" & mgr = 1423
```

Example 4: Find the names of employees who are either in the Admin department or whose sum of salary and commission exceeds $10,000.

```
          select name
          from emp
          where dept = "Admin" ¦ sal + comm > 10000
```

It is possible to specify more complex table look-up operations by using a select-from-where block as the last component of a term in the where clause. This indicates that the comparison specified in the term is to be performed for every value selected by the inner block. All inner select-from-where blocks must be delimited by braces ({}).

Example 5: Find all items sold by departments located on the second floor.

```
          select item
          from sales
          where dept = {select dept
                        from loc
                        where floor = 2}
```

One can apply set functions to the results of a select-from-where block, as shown by the following examples. Allowable set functions are: min, max, count, avg, sum, and user-defined functions. These are discussed in the section, Builtin Functions. User-defined functions are discussed in the description of the declare request.

Example 6: Find the average salary of employees in the shoe department.

```
          avg {select sal
               from emp
               where dept = "Shoe"}
```

Example 7: Find all employees whose salary is greater than that of any employee in the shoe department.

```
          select name
          from emp
          where sal > max {select sal
                           from emp
                           where dept = "Shoe"}
```

A select clause can also contain an arithmetic expression, as shown in the following example.

Example 8: Find each employee in the shoe department, together with his deviation from the average salary of that department.

```
select name sal - avg {select sal
                       from emp
                       where dept = "Shoe"}
from emp
where dept = "Shoe"
```

Set operations can be applied to the results of select-from-where blocks. In LILA the set operations are union, differ, and inter, which correspond to the union, difference, and intersection operations as normally defined. That is, the union of two sets consists of all items which belong to one or both of the sets. The intersection of two sets consists of those items belonging to both sets. The difference of two sets consists of those items which belong to the first set, but not to the second. For example, assume set A contains the letters a, b, and c; while set B contains the letters c, d, and e. Then A union B is a, b, c, d, and e; A inter B is c; and A differ B is a and b, while B differ A is d and e.

Example 9: Find those items which are supplied by Levi and sold in the men's department.

```
select item
from supply
where supplier = "Levi"

inter

select item
from sales
where dept = "Men"
```

It is possible to nest select-from-where blocks in order to specify quite complex selection criteria.

Example 10(a): Find the total volume of items of type A sold by departments on the second floor.

```
sum {select vol
     from sales
     where item = {select item
                   from class
                   where type = "A"}
     & dept = {select dept
               from loc
               where floor = 2}}
```

It is also possible to bypass the block notation and use table names to qualify column names (including *) within the select clause and where clause. This qualification is accomplished by prefixing a column name with a table name followed by a dot (.). Using this approach, the previous expression becomes the following.

Example 10(b):

```
          sum {select vol
               from sales
               where sales.item = class.item & class.type = "A"
                 & sales.dept = loc.dept & loc.floor = 2}
```

Finally, it is possible to specify variables which assume rows of a designated table as values. In certain complex queries, such row designators are required to resolve ambiguity. A row designator is associated with a table by adding a prefix to the table name in the from clause. The prefix consists of the row designator name followed by a colon (:). Several row designators may be associated with a single table. The row designator is used in the select clause and where clause like a table name to qualify a column name. Row designators are global within a LILA expression. Therefore, a row designator which has been associated with a table in one select-from-where block can be referenced in another such block by associating the row designator with a table name of * in the from clause of the referencing block.

Example 11(a): For all employees who earn more than their managers, select the employee's name and that of his manager.

```
          select x.name y.name
          from x:emp y:*
          where x.mgr = {select y.emp_no
                         from y:emp
                         where x.sal > y.sal}
```

Alternatively, experienced users may wish to forego the block notation and specify the same query as follows.

Example 11(b):

```
          select x.name y.name
          from x:emp y:emp
          where x.mgr = y.emp_no & x.sal > y.sal
```

The preceding description is an introduction to the basic features of LILA. The information in this section is sufficient to allow the reader to formulate LILA expressions to satisfy a large class of data selection requirements. However, users should read the section, Syntax and Semantics of LILA, for a precise description of the complete LILA capabilities.

LINUS Requests

The following is a summary of LINUS requests arranged in functional order. The remainder of this section contains a detailed description of each request ordered alphabetically.

open, o
: opens a specified data base or data submodel for processing by LINUS.

declare, dcl
: allows the user to declare user-written functions for later invocation within LILA expressions.

set_scope, ss
: defines the current scope of access within a shared data base.

lila
: invokes the data selection mechanism to process the LILA expression and to select the specified data for manipulation by a subsequent LINUS request.

print, p
: specifies that the selected data is to be retrieved and printed on the terminal in default format.

write, w
: specifies that the selected data is to be retrieved and written to a file in the storage system or to a formatted report.

set
: specifies that the selected data is to be retrieved, and that the retrieved values are to be assigned to the designated variables.

modify, m
: specifies that the selected portion of the data base is to be modified.

delete, d
: specifies that the selected data is to be deleted from the data base.

define_temp_table, dtt
: specifies that the selected data is to form a new, temporary table, known only to the process, but which can be accessed by the process for retrieval in the same manner as data base tables.

store, s
adds new rows to specified tables in the data base.

del_scope, ds
deletes all or a portion of the current scope of access in a shared data base.

invoke, i
executes the requests in the designated LINUS macro segment.

execute, e
passes the remainder of the command line to the Multics command processor for execution.

close, c
closes the currently open data base, making it unavailable to the process until it is again opened.

help, h
provides information on designated LINUS requests to the user at the terminal.

quit, q
terminates a LINUS session.

<u>Request</u>: close, c

This request closes the currently open data base, making it unavailable to the process until it is again opened.

<u>Usage</u>

```
     close
```

Request: declare, dcl

This request allows the user to declare a non-standard function which may be invoked in a subsequent LILA expression. A non-standard function is any function not included in the section, Builtin Functions, and may be user-written or may be provided by the local installation. Two types of functions may be declared: set functions which operate on multiple sets of values, e.g. sum {...}, and scalar functions which operate on one occurrence of a set of values.

Usage

```
     declare fn_name fn_type -control_args
```

where:

1. fn_name
   is the name of the function being declared. The fn_name must be the name of an object segment which can be found using the search rules currently in effect.

2. fn_type
   is the type of the function being declared. Two types are permitted, "set" or "scalar". A set type function operates on multiple sets of selected values, whereas a scalar type function operates on one set of specified values. An example of a set function type is:

   ```
             avg {select salary
                  from emp}
   ```

   while a scalar function example would be:

   ```
             substr (name 1 5)
   ```

3. control_args
   are two arguments, both of which must be specified.

   -input_type declarations, -it declarations
   specifies the type declarations of all input arguments to the function, in the order in which the arguments are specified in the LILA expression. Type declarations are specified in PL/I syntax. Each declaration is assumed to be one character string. Thus, if a declaration contains spaces, it should be enclosed in quotes.

-return_type declaration, -rt declaration
: specifies the type declaration of the return value. Return_type declarations are specified in the same manner as input_type declarations, with the additional type of "logical" allowed. A logical return_type indicates that the function is a truth-valued function, returning the values true or false.

Notes

Input type declarations may contain a * length, indicating that lengths are to be determined from data descriptions within the data base. Any data conversions necessary to resolve differences between the data base and the functions are performed automatically by LINUS according to PL/I conversion rules. The user must refer to the function documentation to determine the appropriate type declarations.

Scalar functions can accept as input column values from only one table, if no row designators are used. If row designators are specified, column names must all be qualified with the same row designator.

Several builtin functions are provided as a standard part of LINUS. See the section, Builtin Functions, for a description of these functions. It is not necessary to declare builtin functions. If a declared function has the same name as a builtin function, the declared function, rather than the builtin function, will be invoked when the function name is referenced.

Example

Find the department average sales volume of all items made of cotton. Assume that the item code contains encoded information indicating the material of which an item is made, and that the user-defined scalar function, material, returns this information. Also assume that there is a user-defined set function, dept_avg, which calculates the desired average, which is the total volume divided by the number of departments. (The "end" following the LILA expression merely serves to indicate the termination of the expression.)

```
declare material scalar -input_type "fixed decimal (8)"
     -return_type "character (12)"
declare dept_avg set -input_type "character (*)" "fixed
     binary (17)" -return_type "fixed binary (17)"
lila
     dept_avg {select dept vol
```

```
                    from sales
                    where material (item) = "cotton"}
          end
```

Request: define_temp_table, dtt

This request causes the selected data to be placed into a temporary table which can then be referenced as any other table in the data base for retrieval purposes. This feature is useful from an efficiency standpoint, since multiple retrievals of the same data can be avoided. This request must immediately follow a lila request.

Usage

```
define_temp_table table_name key_columns
```

where:

1. table_name
   is the name of the temporary table. Subsequent references to this table must use this name. If a temporary table of this name already exists, it will be redefined.

2. key_columns
   are one or more column names specified in the associated LILA select clause which are to become key columns in the temporary table. Key columns uniquely determine the rows of the temporary table, i.e. the concatenation of the values of all key columns must be unique for each row of the temporary table. Duplicates are automatically eliminated.

Notes

The select clause of a LILA expression associated with a define_temp_table request cannot contain an expression. Only column names (qualified or unqualified, including *) are allowed.

All key columns must be explicitly specified in the associated select clause, i.e. a key column cannot be one of those specified by a *.

Temporary tables cannot be updated, but can be accessed for retrieval only.

Normally, a temporary table is created for the purpose of simplifying LILA expressions when data is to be selected from several tables in the data base.

<u>Example</u>

If it were necessary to retrieve information from the department store data base about employees based upon the floor on which they are located, the following temporary table would be useful.

emp_loc (<u>name</u>, emp_no, mgr, sal, comm, floor)

Such a temporary table would be created by:

```
lila
     select name emp_no mgr sal comm floor
     from emp loc
     where emp.dept = loc.dept
end
define_temp_table emp_loc name
```

The LILA expression necessary to find the average salary of all employees located on the second floor would then be:

```
avg {select sal
     from emp_loc
     where floor = 2}
```

as opposed to the following, if the temporary table were not available:

```
avg {select sal
     from emp
     where dept = {select dept
                   from loc
                   where floor = 2}}
```

Request: del_scope, ds

This request deletes all or a portion of the scope of access declared by a previous set_scope command. This request is applicable only for shared (non-exclusive) opening modes.

Usage

```
     del_scope table_name1 {permit_ops1 prevent_ops1 ...
          table_namen permit_opsn prevent_opsn}
```

where:

1. table_namei
   is the name of a non-temporary table within the data base for which all or a portion of the scope of access is to be deleted. If table_name1 is a *, then no additional arguments need be specified, and all of the user's current access scope is deleted.

2. permit_opsi
   is a character string indicating which currently permitted operations are to be deleted from the access scope.

3. prevent_opsi
   is a character string indicating which of the operations currently being prevented for other processes can be deleted from the access scope.

Note

See the description of the set_scope request for a definition of the operation codes, and for a more detailed discussion of the scope mechanism.

Examples

Remove modify permission for the employee table and allow other processes to perform store, modify, and delete operations on it.

```
          del_scope emp m smd
```

Delete all of the current scope of access.

```
del_scope *
```

Request: delete, d

This request deletes selected rows from a single table within the data base. The data base must be open for update or exclusive_update and, if open for update, the affected table must be within the scope of access for delete. This statement must immediately follow a lila request.

Usage

```
delete
```

Notes

The select clause of the associated LILA expression must specify columns from only one table, and all columns from that table must be specified (use of * is recommended). The affected table cannot be a temporary table.

Example

Joe Smith has just been fired. Delete his employee record.

```
lila
     select *
     from emp
     where name = "Smith, Joe"
end
delete
```

Request: execute, e

This request is used to invoke the Multics command processor without exiting from LINUS. Whenever an execute request is recognized, the remaining characters in the request line are passed to the Multics command processor. The execute request can be followed by any legal Multics command line. However, the user should not invoke LINUS while in LINUS since LINUS is not recursive.

Usage

```
     execute command_line
```

where command_line is any legal Multics command line.

Note

This request allows the user to make full use of Multics facilities external to LINUS. Specifically, data may be retrieved and written to a Multics segment by the write request. Then a Report Generator Language (RGL) object segment may be invoked via the execute request to create a formatted report from the retrieved data. Other frequently used facilities are text editors and the sort command.

Example

Retrieve the name, department, and salary of every employee, and create a formatted report containing this information. Assume that an RGL object segment, emp_report, will create the desired report.

```
          lila
               select name dept sal
               from emp
          end
          write emp_data
          execute emp_report emp_data
```

Request: help, h

This request prints information regarding a designated LINUS feature on the terminal. The information available via the help request is concise reference material and is not intended to be tutorial in nature.

Usage

```
     help {feature}
```

where feature is an optional argument designating a specific feature or request about which information is desired. If omitted, a list of features for which information is available is printed.

Examples

Print a list of features for which information is available.

```
     help
```

Print information about the write request.

```
     help write
```

Request: invoke, i

This request specifies that the requests contained in the designated macro segment are to be executed. Arguments may optionally be passed to the macro. This feature provides the capability to invoke a pre-defined series of LINUS requests.

Usage

```
     invoke macro_path {optional_args}
```

where:

1. macro_path
   is the pathname of the ASCII segment containing the LINUS macro. If macro_path does not have a suffix of linus, then one is assumed. However, the suffix linus must be the last component of the name of macro_path.

2. optional_args
   are character strings to be substituted for special strings in the macro segment.

Notes

Upon acceptance of the invoke request, the macro segment is read and executed, line-by-line. Argument substitution also takes place on a line-by-line basis, after the line is read and prior to its execution. After all lines in the macro segment have been processed, LINUS waits for the user to type further requests on the terminal. See the section, Macro Facility, for a complete description of the LINUS macro capability.

Example

Execute the requests contained in the segment get_salary.linus, passing the argument "Smith, John".

```
          invoke get_salary "Smith, John"
```

Request: lila

This request invokes the LILA processor. All text following the "lila" and prior to the "end" is assumed to be a LILA expression. A lila request must immediately precede all print, set, write, modify, delete, and define_temp_table requests.

Usage

```
     lila LILA_expression end
```

where LILA_e:)ression is an expression in LILA which selects data from the data base. See the sections Selection Language and Syntax and Semantics of LILA for a detailed description of a LILA expression. The words lila and end serve as delimiters for the LILA_expression.

Example

See the examples for the print, set, write, modify, delete, and define_temp_table requests.

<u>Request</u>: modify, m

This request modifies selected data in the data base. Data to be modified must be contained within one table, and key columns cannot be modified. The data base must be open for update or exclusive_update. If open for update, the table being updated must be within the current access scope for the modify operation. New values may be specified within the request line, or they may be entered interactively, in response to LINUS prompting. In both cases, the user is asked to verify the new values before the modification takes place, unless the verification mode is explicitly turned off. This request must immediately follow a lila request.

<u>Usage</u>

```
     modify {column_values} {-control_arg}
```

where:

1. column_values
   are optional arguments and, if present, specify the new values that are to replace the current values of the data selected by the associated LILA expression. The column_values must be specified in the same order that the associated column names are listed in the select clause. If not present, LINUS will request the column_values individually by name.

2. control_arg
   may be either -brief or -bf which specifies that verification of column_values is not to be done. If not present, LINUS will print a list of selected column names, together with the column_values as entered by the user, and request that the user verify the correctness of the column_values before the modification operation proceeds. If the verification is negative, the modification does not take place. The user may reenter the modify request without again specifying the associated LILA expression.

<u>Notes</u>

New column_values may be specified in two forms: as constants or LINUS variables which have previously been set, or as arithmetic expressions combining constants, LINUS variables, and column names specified in the select clause of the associated

LILA expression.

The select clause of the associated LILA expression must specify columns from only one table, and only non-key columns may be selected. The select clause associated with a modify request may not contain arithmetic expressions, but is restricted to simple or qualified column names.

Examples

Give every employee a 10 per cent raise.

```
lila
     select sal
     from emp
end
modify sal + .10 * sal

sal = sal + .10 * sal
OK?    yes
```

Al Jones has transferred to the shoe department. Update his employee record to indicate his new department and manager.

```
lila
     select dept mgr
     from emp
     where name = "Jones, Al"
end
modify

dept?    Shoe
mgr?    1234

dept = Shoe
mgr = 1234
OK?    yes
```

Update the data base to indicate that the shoe department has moved to the third floor.

```
lila
     select floor
     from loc
     where dept = "Shoe"
end
modify 3 -brief
```

<u>Request</u>: open, o

This request opens a specified MRDS data base with the designated opening mode. The data base may be designated either by the pathname of the data base itself, or by the pathname of a data submodel associated with the data base. Only one data base may be open at any given time.

<u>Usage</u>

```
open data_path mode
```

where:

1. data_path

   is the pathname of a MRDS data base or of a data submodel associated with a MRDS data base. A data submodel is a user's view of the data base which may differ from the actual data base definition. See the MRDS Reference Manual for a more detailed discussion of data models and data submodels.

2. mode

   is the usage mode for which the data base is to be opened. Modes may be specified either by their full names or by their abbreviations. The folowing are valid opening modes.

   retrieval, r
   : indicates that the user wishes only to retrieve data from the data base and will allow concurrent access, for both update and retrieval, by other users.

   update, u
   : indicates that the user wishes to both retrieve and update information in the data base and will allow concurrent access, for both update and retrieval, by other users.

   exclusive_retrieval, er
   : indicates that the user wishes only to retrieve data from the data base, but that concurrent access by other users for update is to be prohibited.

   exclusive_update, eu
   : indicates that the user wishes to both retrieve and update information in the data base, and that no concurrent access by other users is to be permitted.

Notes

If the designated data base is already open for another user in a mode that conflicts with the mode designated in the open request, the open request will be denied.

Several data bases may be opened during a LINUS session, so long as each one is closed prior to opening the next.

Example

Open the department store data base for non-exclusive retrieval.

```
open dept_store retrieval
```

Request: print, p

This request specifies that selected data is to be retrieved and printed on the user's terminal. The selected columns are printed side-by-side with optional column headers. The user may specify that a limit be placed on the number of rows to be printed. This request must immediately follow a lila request.

Usage

```
print {-control_args}
```

where control_args may be one or more of the following.

-max n
: where n is a positive integer, specifies that no more than n rows of information are to be printed. If not present, all retrieved data is printed.

-no_header, -nhe
: specifies that column headers are not to be printed. If not present, column headers consisting of column names are printed if columns are selected. If an expression is selected, the column header will be f(name), where name is the first column name in the expression.

Notes

The columns are printed side-by-side. The width of each column is determined from the data descriptions in the data base. Each column is separated from the next by one blank. There is no pagination.

Example

Print the names of all employees in the shoe department, together with the sums of their salaries and commissions.

```
lila
    select name sal + comm
    from emp
    where dept = "Shoe"
end
print
```

name	f(sal)
Smith, John	10000
Jones, Al	12000
Anderson, Carol	8000
Johnson, Betty	11000

Request: quit, q

This request terminates the LINUS session. If a data base is open at the time of this request, it is automatically closed.

Usage

```
     quit
```

Request: set

This request specifies that selected data is to be retrieved and the retrieved values assigned to designated LINUS variables. This capability allows information obtained from one retrieval to be used in subsequent data base accesses. The LINUS variables can also be passed as arguments to LINUS macros.

Usage

```
     set variable_list
```

where variable_list is a list of one or more variable names.

Notes

A variable name is an alphanumeric character string, from one to 32 characters in length, which begins with an exclamation mark (!). The underscore (_) and hyphen (-) may also be included, but the exclamation mark may not appear elsewhere in the name. The specification in a set request is the only declaration required. If the same variable is specified in several set requests, its value is reset in each of those set requests. Variable names and values are preserved across data base openings and closings.

It should be noted that variables specified in the set request are unrelated to row designators in LILA.

Variables represent character data. The length of the character string represented by a variable is dependent upon the data base description of the data item being assigned to the variable.

Retrieved data is assigned to variables in the variable_list in the order that it is retrieved. Retrieval ceases when all selected data is exhausted or when all variables in the variable_list are exhausted, whichever occurs first.

Variable names are global within a LINUS session, i.e. like variable names occurring in different LINUS macros will refer to the same variable, if the macros are used in the same LINUS session.

Example

Find the names and total compensations for those employees whose total compensations are more than 50 per cent below average or are more than 50 per cent above average.

```
          lila
               avg {select sal + comm
                    from emp}
          end
          set !avg_comp
          lila
               select name sal + comm
               from emp
               where sal + comm < .5 * !avg_comp
                 | sal + comm > 1.5 * !avg_comp
          end
          print
```

Request: set_scope, ss

This request allows the user to define his current scope of access to the data base for non-exclusive opening modes. This request and the del_scope request are the means through which the user defines his requirements to the LINUS concurrent access control mechanism. Every table which the user wishes to access for a given period must be included within the user's scope of access for that same period.

For every table to be included in the current scope, the user specifies the types of access he will require, and also those types of access which are to be prohibited to other users. The scope of access is a dynamic entity, and may be varied to reflect the user's changing requirements during the life of a LINUS session. In order to prevent deadlock situations, however, the current scope must be set to null with the del_scope request prior to issuing a set_scope request.

Usage

```
set_scope table_name1 permit_ops1 prevent_ops1 {...
     table_namen permit_opsn prevent_opsn}
```

where:

1. table_name*i*
   is the name of a non-temporary table within the data base which is to be included in the current scope of access.

2. permit_ops*i*
   is a character string indicating which types of data base operations are to be permitted the user for the corresponding table. The character string is the concatenation of the codes for all operations to be permitted. See notes below for a description of the operation codes.

3. prevent_ops*i*
   is a character string similar to that for permit_ops*i* indicating which types of data base operations are to be denied other users for the corresponding table.

Notes

Codes for operation types to be permitted or prevented are as follows:

Operation	Code
retrieve	r
store	s
delete	d
modify	m
null	n

It is recommended that users declare the minimum access scope necessary for any given operation, and that the scope be maintained for only as long as it is needed. Declaration of unnecessarilly large scopes is discouraged, as other users may be needlessly locked out of the data base.

The set_scope request will be denied if the user currently has a non-null scope in force. Therefore, all of the user's access scope must have been deleted with a del_scope request prior to issuing a set_scope. The set_scope request must then specify the entire scope of access required by the user for a block of operations. This is in contrast to the del_scope request, where portions of the current scope may be deleted. If another user has a conflicting scope in force, the set_scope request will also be denied.

Example

Jim Jones, the manager of the shoe department, has retired and will be replaced by Al Smith. Update the employee table to reflect these changes, while ensuring that no other users access inconsistent data.

```
set_scope emp r n
lila
     select emp_no
     from emp
     where name = "Smith, Al"
end
set !smith_no
lila
     select emp_no
     from emp
     where name = "Jones, Jim"
end
set !jones_no
del_scope *
set_scope emp dm rdms
lila
     select mgr
```

```
          from emp
          where mgr = !jones_no
     end
     modify !smith_no -brief
     lila
          select *
          from emp
          where name = "Jones, Jim"
     end
     delete
     del_scope *
```

Request: store, s

This request adds new rows 'ɔ a designated table in the data base. The data base must be open for update or exclusive_update. If open for update, the table being stored must be within the current access scope for the store operation. Values being stored may be specified in one of three ways: directly within the request line, interactively in response to LINUS prompting, or by placing the values in a Multics file and supplying the pathname as a control argument in the store request line. Using the first two methods, only a single row may be stored with one store request, whereas the third method (file input) allows the storing of multiple rows. Also, if the new row is being entered from the terminal (as opposed to file input), the user has the option of verifying the values prior to their being stored into the data base.

Usage

```
store table_name {column_values} {-control_args}
```

where:

1. table_name
   is the name of the table to which rows are being added. This must be the name of a non-temporary table.

2. column_values
   are optional arguments and, if present, specify the column values comprising the new row being added. The column_values must be specified in the same order that the corresponding columns appear in the data base or the data submodel, whichever is applicable. Also, exactly one value must be specified for every column defined in the data base or data submodel.

3. control_args
   may be one or more of the following.

   -brief, -bf
   specified that verification of column_values is not to be done. If not present, and if the -file control argument is not present, LINUS will print a list of column names, together with the column_values entered by the user, and request that the user verify the correctness of the column_values before the store operation proceeds. if the verification is negative, the store does not take place, and the user must

reenter the store request.

-file pathname, -f pathname
: specifies that the column_values are to be taken from the Multics file designated by pathname. This pathname must designate a Multics file suitable for processing by vfile_ in the stream_input opening mode. See notes for a detailed description of the input file.

-delimiter char, -dm char
: specifies that each column_value, in the file specified via -file, is separated from the next by the character, char. This control argument has meaning only if specified together with -file. If not present, each column_value is assumed to be delimited by one or more blanks.

Notes

If column_values are not present in the request line and -file is not specified, then LINUS will request each column_value individually by name.

If -file is specified, the input file may contain column_values for more than one row. The input for each row is terminated by a newline character. In all cases, column_values are separated by blanks unless another delimiter is specified via -delimiter.

Examples

Add a new supplier to the supply table.

```
store supply Acme 10 200

supplier = Acme
item = 10
vol = 200

OK?  yes
```

Another way of performing the above operation is:

```
store supply -brief

supplier?   Acme
item?   10
vol?   200
```

Several rows could be added to the supply table by first creating the following file with a text editor:

```
Acme,10,200
XYZ,12,150
J. Smith,10,100
```

and then entering the following request:

```
store supply -file supply_file -delimiter ,
```

<u>Request</u>: write, w

This request specifies that the selected data is to be retrieved and written to the specified Multics file. The output file is a text file created by vfile_ in the stream_output mode. If the file already exists, it may optionally be extended, although normally it would be truncated. An option to invoke a RGL object module as an I/O appendage to the report_ I/O Module is also provided. This latter option provides the capability to directly create formatted reports from selected data. This request must immediately follow a lila request.

<u>Usage</u>

```
     write {outfile} {-control_args}
```

where:

1. outfile

   is optional and, if present, is the pathname of a Multics file into which the selected data is to be written. If the file does not currently exist, it will be created. If the file currently exists, it will be truncated unless -extend is also specified. If this argument is not present, -report must be specified.

2. control_args

   may be one or more of the following.

   -extend
   specifies that if the outfile exists, it is to be added to, rather than truncated.

   -delimiter char, -dm char
   specifies that each selected value is to be delimited by the character, char, in the outfile. If not present, each selected value is delimited by one blank.

   -report arg_string, -rp arg_string
   specifies that the data is to be written using the report_ I/O Module with the argument string, arg_string. Arg_string is a character string which must begin with the name of the desired RGL object module, and must also contain any arguments required by the RGL object module. The output switch is attached via report_ and is opened in the stream_output mode. Each set of selected values is

written as a line through report_. Each selected value is delimited as specified by -delimiter. If outfile is not specified, -report must be present. If -report is not present, outfile must be specified.

Notes

The output file is a text stream file created by vfile_. Each set of selected values is delimited by a newline character. The output file is suitable for processing by a text editor, as well as other Multics facilities which process ASCII text files.

Examples

Create a text file consisting of the name and salary of every employee.

```
lila
     select name sal
     from emp
end
write salary_file
```

Create a formatted report containing the name, department, and salary of every employee. Assume that the RGL object segment, emp_report, will create the desired report.

```
lila
     select name dept sal
     from emp
end
write -report emp_report
```

Builtin Functions

The following is a list of builtin functions available in LINUS. Each of the functions is subsequently described in detail.

```
abs          max
after        min
avg          mod
before       reverse
ceil         round
concat       search
count        substr
floor        sum
index        verify
```

FUNCTION: abs

This is a scalar arithmetic function and a reference to this function has the form:

```
abs (X)
```

The result of this function is the absolute value of X. X must be a numeric data item. If X is real, the result has the same data definition as X. If X is complex then

```
abs (X + Yi) = +sqrt (X ** 2 + Y ** 2)
```

FUNCTION: after

This is a scalar string function and a reference to this function has the form:

```
after (S1 S2)
```

The result is that portion of S1 which occurs after the leftmost occurrence of S2 within S1. If S2 is a null string, the result is S1. If S2 does not occur within S1, the result is a null string. For example:

```
after ("abcde" "bc") = "de"
after ("abcde" "") = "abcde"
after ("abcde" "f") = ""
after ("10101"b "10"b) = "101"b
```

FUNCTION: avg

This is an arithmetic set function and a reference to this function has the form:

```
avg {select X
     from ...}
```

The result is the average (mean) of all X values selected. For example:

```
avg {select sal
     from emp
     where dept = "Shoe"}
```

is the average salary of all employees in the shoe department.

FUNCTION: before

This is a scalar string function and a reference to this function has the form:

```
before (S1 S2)
```

The result is that portion of S1 which occurs before the leftmost occurrence of S2 within S1. If S2 is a null string, the result is a null string. If S2 does not lie within S1, then the result is S1. For example:

```
before ("abcde" "bc") = "a"
before ("abcde" "") = ""
before ("abcde" "f") = "abcde"
before ("10101"b "10"b) = ""b
```

FUNCTION: ceil

This is a scalar arithmetic function and a reference to this function has the form:

```
ceil (X)
```

where X must be real. The result is the smallest integer (I) such that

```
I >= X
```

For example:

```
ceil (20.5) = 21
ceil (-14.6) = -14
ceil (12) = 12
```

FUNCTION: concat

This function is a scalar string function and a reference to this function has the form:

```
concat (S1 S2)
```

The result is the concatenation of S1 and S2. For example:

```
concat ("abc" "de") = "abcde"
concat ("101"b "01"b) = "10101"b
```

FUNCTION: count

This is an arithmetic set function and a reference to this function has the form:

```
count {select X1 X2 ...
       from ...}
```

The result is the number of sets of Xi which are selected. For example:

```
count {select name
       from emp
       where dept = "Shoe"}
```

is the number of employees in the shoe department.

FUNCTION: floor

This is an arithmetic scalar function and a reference to this function has the form:

```
floor (X)
```

where X is real. The result is the largest integer (I) such that

```
I <= X
```

For example:

```
          floor (20.5) = 20
          floor (-14.6) = -15
          floor (12) = 12
```

FUNCTION: index

This is a scalar string function and a reference to this function has the form:

```
          index (S1 S2)
```

The result is an integer that is the position of the beginning of the leftmost occurrence of S2 within S1. If S2 is not in S1 then the result is 0. If S2 is a null string, the result is 0. For example:

```
          index ("abcde" "bc") = 2
          index ("abcde" "f") = 0
          index ("abcde" "") = 0
```

FUNCTION: max

This is an arithmetic set function and a reference to this function has the form:

```
          max {select X
               from ...}
```

The result is the largest X value selected. For example:

```
          max {select sal
               from emp
               where dept = "Shoe"}
```

is the highest salary paid to any employee in the shoe department.

FUNCTION: min

This is an arithmetic set function and a reference to this function has the form:

```
          min {select X
               from ...}
```

The result is the smallest X value selected. For example:

```
          min {select sal
               from emp
               where dept = "Shoe"}
```

is the lowest salary paid to any employee in the shoe department.

FUNCTION: mod

This is an arithmetic scalar function and a reference to this function has the form:

```
          mod (X Y)
```

where X and Y are real. The result is X modulus Y, i.e.

```
          if Y ^= 0 then mod (X Y) = X - Y * floor (X / Y)
          if Y = 0 then mod (X Y) = X
```

For example:

```
          mod (42 5) = 2
          mod (129.2867 25) = 4.2867
          mod (10 0) = 10
```

FUNCTION: reverse

This is a scalar string function and a reference to this function has the form:

```
          reverse (S)
```

The result is a string which is the reverse of the value of S. For example:

```
          reverse ("abcde") = "edcba"
          reverse ("a") = "a"
          reverse ("") = ""
          reverse ("10110"b) = "01101"b
```

FUNCTION: round

This is a scalar arithmetic function and a reference to this function has the form:

```
          round (X Q)
```

The result is a rounding of the value of X. When a value is rounded to n digits, the digits after the nth digit are dropped, and the nth digit is increased by 1 if the (n+1)th digit is 5 or greater for decimal, or 1 for binary. If X is float, then Q must be positive and the mantissa is rounded to Q digits. If X is fixed, it is rounded to a value that has Q fractional digits. For complex values, the function is defined by:

```
round (X + Yi Q) = round (X Q) + round (Y Q)i
```

For example:

```
round (183.629e6 4) = 183.6e6
round (183.629 2) = 183.63
round (183.629 -1) = 180
round (21.56 + 6.21i 0) = 22 + 6i
```

FUNCTION: search

This is a scalar character string function and a reference to this function has the form:

```
search (C1 C2)
```

The result is an integer value that is the position in C1 of the leftmost occurrence of any character contained in C2. If C1 does not contain any character in C2, the result is 0. For example:

```
search ("abcde" "b") = 2
search ("abcde" "") = 0
search ("abcde" "f") = 0
search ("abcde" "be") = 2
```

FUNCTION: substr

This is a scalar string function and a reference to this function has one of the forms:

```
substr (S I J)
substr (S I)
```

The result is that portion of S which begins with the Ith character and has length J if J is present, or is that portion of S which begins with the Ith character and continues to the end of S if J is not present. For example:

```
substr ("abcde" 3 2) = "cd"
substr ("abcde" 3 0) = ""
substr ("abcde" 3) = "cde"
```

```
          substr ("10101"b 3) = "101"b
```

FUNCTION: sum

This is an arithmetic set function and a reference to this function has the form:

```
          sum {select X
               from ...}
```

The result is the total of all selected values. For example:

```
          sum {select vol
               from sales
               where dept = "Shoe"}
```

provides the total sales volume of the shoe department.

FUNCTION: verify

This is a character string scalar function and a reference to this function has the form:

```
          verify (C1 C2)
```

The result is an integer value that is the position of the first character of C1 that does not occur in C2. When C1 contains only characters that are in C2, the result is 0. For example:

```
          verify ("xyz" "abc") = 1
          verify ("xyz" "xyz") = 0
          verify ("abcde" "cba") = 4
```

Macro Facility

LINUS provides the capability to execute a series of requests contained in a text segment. Such a segment is referred to as a LINUS macro segment. The name of a LINUS macro segment must have a suffix of linus.

A LINUS macro may be invoked in one of two ways: (1) via the linus command line or (2) via the LINUS invoke request. Invocation via the linus command line is as follows:

```
          linus macro_path arg1 ... argn
```

which is equivalent to the following sequence:

```
          linus
          invoke macro_path arg1 ... argn
```

A LINUS macro segment contains a series of LINUS requests in the same format as if they were entered at the terminal. Comments may appear in a LINUS macro segment in the same manner that they may appear in a PL/I source segment. It is possible to specify arguments to the LINUS macro in a method analogous to the specification of arguments to a Multics exec_com. In a LINUS macro, strings of the form %i% are interpreted as dummy arguments and are replaced by the corresponding optional_args in the invoke request or in the linus command line. For example, optional_arg1 is substituted for the string %1% and optional_arg10 is substituted for the string %10%. Substitutions are also made within quoted strings. If a % is to be included in a string, %% should be specified.

The following is an example of a LINUS macro which prints the sales volume, given a department name and item code.

```
          open dept_store retrieval /* open data base */
          set_scope sales r n /* allow read only, no prevents */
          lila                /* specify the data */
               select vol
               from sales
               where dept = "%1%" & item = %2%
          end
          print -no_header       /* no need for header */
          del_scope *         /* clean up */
          close
          quit
```

Assume the above macro resides in the segment volume.linus. Then, in order to obtain the sales volume for item 20 in the shoe department, the user types:

```
          linus volume shoe 20
```

and the resulting where clause reads:

```
          where dept = "shoe" & item = 20
```

Syntax and Semantics of LILA

The following is a formal syntax for LILA.

```
<lila_expr> ::= <set_value> ¦ <lila_set>

<set_value> ::= <set_fn> {<lila_set>}

<set_fn> ::= <set_builtin> ¦ <user_set_fn>

<lila_set> ::= <lila_block> ¦ <lila_set> <set_op> <lila_block>
             ¦ {<lila_set>}

<set_op> ::= union ¦ inter ¦ differ

<lila_block> ::= select <select_list> from <from_list>
               ¦ select <select_list> from <from_list>
                 where <conditional>

<select_list> ::= * ¦ <select_item_list>
                ¦ dup <select_item_list>
                ¦ unique <select_item_list>

<select_item_list> ::= <select_item>
                     ¦ <select_item_list> <select_item>

<select_item> ::= <table_name>.* ¦ <row_desig>.* ¦ <expr>

<expr> ::= <column_spec> ¦ <scalar_fn> (<arg_list>)
         ¦ <expr> <arith_op> <arithmetic_constant>
         ¦ <expr> <arith_op> <set_value>
         ¦ <expr> <arith_op> <expr>
         ¦ (<expr>)

<column_spec> ::= <column_name>
                ¦ <table_name>.<column_name>
                ¦ <row_desig>.<column_name>

<scalar_fn> ::= <scalar_builtin> ¦ <user_scalar_fn>

<arg_list> ::= <arg> ¦ <arg_list> <arg>

<arg> ::= <expr> ¦ <constant> ¦ <set_value>

<arith_op> ::= + ¦ - ¦ * ¦ /

<from_list> ::= <table_name> ¦ <table_list>

<table_list> ::= <row_tab_pair> ¦ <table_list> <row_tab_pair>

<row_tab_pair> ::= <row_desig>:<table_name> ¦ <row_desig>:*
```

```
<conditional> ::= <term> ¦ <conditional> <bool_op> <term>
                ¦ ^(<conditional>) ¦ (<conditional>)

<term> ::= <expr> <rel_op> <atom>

<rel_op> ::= = ¦ ^= ¦ > ¦ < ¦ >= ¦ <=

<bool_op> ::= & ¦ ⊥

<atom> ::= <expr> ¦ <constant> ¦ <set_value> ¦ {<lila_block>}

<constant> ::= <arithmetic_constant>
             ¦ <bit_string_constant>
             ¦ <character_string_constant>
             ¦ <linus_variable>

<linus_variable> ::= !<identifier>

<table_name> ::= <identifier>

<row_desig> ::= <identifier>

<column_name> ::= <identifier>

<user_set_fn> ::= <fn_name>

<user_scalar_fn> ::= <fn_name>
```

A <set_builtin> is one of the set builtin functions described in the section, Builtin Functions. A <scalar_builtin> is one of the scalar builtin functions described in the same section. A <user_set_fn> and a <user_scalar_fn> must be declared according to the specifications contained in the declare request description.

<lila_set>s within a <lila_set> may optionally be grouped by braces {} to explicitly specify the order of evaluation. If not explicitly specified, evaluation proceeds from left to right.

The <set_op>s union, inter, and differ correspond to the set operations union, intersection, and difference respectively.

If the where clause is omitted from a <lila_block>, all rows within the <from_list> qualify.

A <select_list> of * indicates that all column values from the row are to be selected. If the <select_list> is a *, then the <from_list> must be a <table_name>.

A specification of dup within a <select_list> indicates that duplicate sets of selected values are not to be eliminated, whereas a specification of unique indicates that duplicates are to be eliminated. If neither is specified, the default rule will apply. The default is dup if a <set_fn> is to be applied to the selected values, and is unique otherwise.

A <select_item> of <table_name>.* or <row_desig>.* indicates that all columns from the row are to be selected. A <table_name> is the name of a previously defined temporary table or of a table defined within the data base. A <row_desig> is a row designator which has been associated with a <table_name> in a <from_list>.

All <column_spec>s within an <expr> or <arg_list> must refer to column values from the same row.

Items within an <expr> may optionally be grouped by parentheses () to explicitly determine the order of evaluation. If not explicitly specified, all multiplications and divisions are performed before any additions or subtractions. Multiplications and divisions are performed from left to right, as are additions and subtractions.

A <row_tab_pair> is used to specify the association of a row designator with a table. A <row_tab_pair> consisting of <row_desig>:* indicates that the row designator is associated with a table in another select-from-where block.

Items within a <conditional> may optionally be grouped by parentheses to explicitly specify the order of evaluation. If the order is not explicitly specified, the evaluation proceeds from left to right.

The items <arithmetic_constant>, <bit_string_constant>, and <character_string_constant> are as defined in Multics PL/I. An <identifier> is as defined in Multics PL/I with the exceptions that the dollar sign ($) is not allowed, and the hyphen (-) is allowed, so long as it is not the first or last character of the <identifier>. A <fn_name> is the same as the LILA <identifier> except that the hyphen (-) is not allowed.

Writing Non-Standard Functions

This section provides information necessary for the creation of non-standard functions. These functions may be written in any language which can accept and process a standard Multics argument list. It is assumed that these functions will be written by experienced programmers.

Scalar functions will be passed a complete standard Multics argument list containing argument pointers and descriptor pointers for both the input arguments and the return argument. The call is equivalent to:

```
return_val = fn_name (in_arg1, ..., in_argn);
```

Set functions are called a bit differently in that they are called several times and that two procedure entry points are required. The first entry point is the calc entry, which is called one time for each set of selected values. This entry is passed a complete standard Multics argument list containing argument pointers and descriptor pointers for all of the declared input arguments. The purpose of the calc entry point is to calculate (or accumulate) the value for the set function. The call to the calc entry is equivalent to:

```
call fn_name$calc (in_arg1, ..., in_argn);
```

The second entry point of a set function is the assign entry. This entry is called after the calc entry has been called for all sets of selected values. The purpose of the assign entry is to actually assign a return value for the set function. The call to this entry is equivalent to:

```
return_val = fn_name$assign ();
```

It is obvious that this method of operation requires that the set function value is to be calculated in static storage. Hence, the assign entry must also reset the function value after the assignment takes place, so that subsequent calls will operate correctly.

A function return type declaration of logical is implemented as a bit (1) unaligned value which is set to "1"b to indicate true, and "0"b to indicate false.

Bibliography

1. Astrahan, M. M. et al. System R: Relational Approach To Database Management. ACM Transactions on Database Systems. Vol. 1, No. 2, June 1976, pp. 97-137.

2. Chamberlin, D. D. and Boyce, R. F. SEQUEL: A Structured English Query Language. Proc. ACM SIGFIDET Workshop, Ann Arbor, Mich., May 1974, pp. 249-264.