To: Distribution

From: Jeff Broughton

Date: October 27, 1975

Subject: Extensible Command Language for Use on Multics

EXTENSIBLE COMMAND LANGUAGE

(ECL)

The purpose of this document is to describe an extensible command language and command environment for use on the Multics system which functionally incorporates the abilities of the current command processor and its accomplices, abbrev and exec_com, and additionally provides the user with more convenient mechanisms for dealing with the command environment and for creating his own commands.

FEATURES

- o Provides a well endowed, interpretive, procedural language supporting variables, arithmetic, string and logical operations, a powerful control structure including conditionals and iteration, and a condition mechanism.

- o Allows definition of user commands and functions by use of procedures written in the command language itself. These procedures would be partially compiled and as such would execute faster than current exec_com files.

- o Provides a mechanism for the automatic definition of the syntax and semantics of the arguments to a user command (procedure) including error detection and reporting.

- o Supports a number of special data types, e.g. pathname or iocb, that correspond to the things normally manipulated by the user at command level allowing him to deal with them in a high level fashion ignoring the details of the of the supervisor interface.

- o Allows the user to control, through block structure, the environment in which a command executes. The environment may be iteratively changed, as in walk_subtree, or it may be limited for use in a restricted subsystem.

o Provides a means of defining other languages, such as a data compiler like cv_pmf or command subsystems such as a debugger using the command processor to interpret the statements (commands).

METHOD OF DEFINITION

The method of defining the ECL language is the same as used for defining the Multics PL/I language. See Section 1.2 of the *Multics PL/I Language Reference Manual*, Order No. AG94, Rev. 1.

THE INTERPRETATION OF ECL

The ECL language describes a sequence of operations to be performed in terms of an <active unit>:

```
<active unit> ::= <program unit>|<statement unit>

     <program unit> ::= <procedure>

     <statement unit> ::= <executable unit>|<declarative>

          <executable unit> ::= <group>|<on unit>|
                      <for unit>|<data unit>|<independent>
```

A <program unit> describes a subroutine that may be called by other programs.

A <statement unit> describes a single action that may be performed at the direct request of the user. The processing of a <statement list> is directed by the command interpreter, and such processing is said to be performed at command level.

Input containing an <active unit> is read and processed by the translator. A syntactically correct unit is passed to the processor for execution.

The Structure of ECL

BLOCKS

A <block> is the most important syntactic form in the language: it controls the flow of execution, delimits the meaning of names, and controls the environment of execution.

```
<block> ::= <procedure>|<on unit>|<for unit>|<data unit>

<procedure> ::= <procedure statement>[<parameter block>]
                          [<procedure body>]<End statement>
     <procedure body> ::= {<body unit>|
```

```
                                  <Entry statement>[<parameter block>]
        <body unit> ::= <block>|<declarative>|
                                        [<label>]<executable unit>

    <on unit> ::= <On statement>[<parameter block>
                                    [<on unit body>]<End statement>
        <on unit body> ::= <body unit>...

    <for unit> ::= <For statement>[<for unit body>]
                                                   <End statement>
        <for unit body> ::= <body unit>...

    <data unit> ::= <Data statement>[<data unit body>]
                                                   <End statement>
        <data unit body> ::= <body unit>...
```

All of the components of a <block> are said to be contained in that <block>. The components of a <block> that are not also contained in a <block> itself contained in the original, are said to be immediately contained in the original <block>. A <procedure> that is a <program unit> is not contained in any <block>.

GROUPS

A <group> describes a group of <statement>s within which there is an internal flow of execution.

```
    <group> ::= <do group> | <if group>

        <do group> ::= <Do statement>[<statement list>]
                                                   <End statement>
           <statement list> ::= {[<label>]<executable unit>}...

        <if group> ::= <If statement><then part>[<else part>]
           <then part> ::= Then <executable unit>
           <else part> ::= Else <executable unit>
```

The internal flow of execution is determined by the interpretation of the <Do statement> or <If statement>.

Note: an <if group> that is an <executable unit> comprising a <statement unit> may not contain an <else part>.

STATEMENTS

All higher level constructs, e.g. an <active unit> or <block>, are formed from a list of <statement>s each with an optional <prefix>. The <statement>s recognized by the language are:

```
<statement> ::= <independent>|<dependent>|<declarative>|
                                        <descriptive>|<invalid>

    <independent> ::=
        <Call statement> |
        <Let statement> |
        <Exit statement> |
        <Continue statement> |
        <Goto statement> |
        <Interpret statement> |
        <Perform statement> |
        <Signal statement> |
        <Revert statement> |
        <Return statement> |
        <Resignal statement> |
        <null statement>

    <dependent> ::=
        <If statement> |
        <Do statement> |
        <For statement> |
        <Data statement> |
        <On statement> |
        <procedure statement> |
        <Entry statement> |
        <End statement>

    <declarative> ::=
        <scope statement> |
        <Synonym statement> |
        <Environment statement>

    <descriptive> ::=
        <Semantics statement> |
        <keyword spec> |
        <type spec> |
        <value spec> |
        <Group statement> |
        <Multiple  statement> |
        <Select statement> |
        <Form statement>
```

All statements must be terminated by a <newline> or semicolon. The syntax and semantics of individual <statement>s is described in the following section.

Independent statements are those which describe an explicit action to be performed.

Dependent statements are used to build <procedure>s, <on unit>s, and <group>s as described above.

Declarative statements are used to define names in the program, and to establish the rules for resolving a definition.

Descriptive statements are used to form a <parameter block>.

An <invalid> statement is any group of <token>s delimited by a <newline> or semicolon that does not correspond to one of the other types of statements. This includes any group of <token>s that begins with an <identifier> that is not a statement name.

STATEMENT PREFIXES

Statement prefixes are used to name a statement, or to control its meaning within the program:

```
<prefix> ::=
     <label> | <control prefix> | <form prefix> | <parm option>

     <label> ::= <identifier> :

     <control prefix> ::= Then | Else

     <form prefix> ::= Optional | Repeat

     <parm option> ::= Default | Error | From
```

A <label> defines an <identifier> as a name for the following statement. Lexically, it may only appear at he beginning of a <line> containing a <statement> that may begin an <executable unit>: e.g. a <Do statement>, an <On statement>, or an <independent> statement.

A <control prefix> is used to designate the alternative actions in an <If group>. It may only appear on a statement that may begin an <executable unit>.

<form prefix>es and <parm option>s are used in <parameter block>s, and are discussed in that section.

LEXICAL SYNTAX

Each <statement> (and optional <prefix>) is formed from a <line> consisting of a group of one or more basic syntactic units called <token>s. The format of these <token>s describe the conventions needed to input a valid <statement>:

```
     <line> ::= [<token>]...{<newline>|;}

     <token> ::= <string>|<identifier>|<operator>|
                              <option name>|<comment>|<delimiter>
```

```
<string> ::= <quoted string>|<unquoted string>

<quoted string> ::= "[<char>|""]..."
   <char> ::= any ASCII character except "

<unquoted string> ::= <chars>...
```

<Excluding all such strings that are <identifier>s, <operator>s, or <option name>s.

```
   <chars> ::= <letter>|<digit>|≤|≥|=|.|+|-|*|/|^|$|!|&|~|⊥

   <letter> ::= <capital>|a|b|c|d|e|f|g|h|i|j|
                          k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
   <capital> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
                          N|O|P|Q|R|S|T|U|V|W|X|Y|Z
   <digit> ::= 0|1|2|3|4|5|6|7|8|9

<identifier> ::= <capital>[<ichar>...]
   <ichar> ::= <letter>|<digit>|_|$|%

<operator> ::= +|-|*|/|**|^|=|^=|≤|≥|<=|>=|&|⊥|⊥⊥|!|==

<option name> ::= -<letter>[<chars>...]

<comment> ::= /*[<any>...]*/
   <any> ::= any ASCII character

<delimiter> ::= <punctuation>|<white space>

<punctuation> ::= ,|:|(|)|[|]|{|}

<white space> ::= <blank>|<tab>
```

Though not explicitly required by the syntax of statements, all <string>s, <identifier>s, <operator>s, <option name>s, and <comment>s must be separated by one or more delimiters. <comment>s can appear freely in any location, as can <white space> which must be used to satisfy the above restriction if there is no <punctuation> required.

A <line> that contains no <token>s is discarded and replaced by a non-null <line>.

Translation

The process of translation causes the input to be read and matched to the syntax of an <active unit>.

TRANSLATION OF A PROGRAM UNIT

The source of a <program unit> resides in a Multics segment named program-name.ecl.

The translator for a <program unit> reads <line>s of the source file and matches them against the list of defined <statement>s. A <label> may not appear in the input. If an <invalid> statement is encountered, an error is reported and processing continues with the next <line>.

The <statement>s found by the translator must form a complete, syntactically correct <procedure>. If not an error is reported.

If there is no error in the input source, the translator for a <program unit> generates an object segment named program-name containing the <program unit> with entrypoints corresponding to each <procedure statement> or <Entry statement> immediately contained in the outermost <procedure>.

The <program unit> is passed to the processor when one of the entrypoints is invoked.

TRANSLATION OF A STATEMENT UNIT

The source of a <statement unit> is read from the I/O switch user_input, and as such may be directly entered by the user.

The translator for a <statement unit> reads <line>s of input and matches them against the list of defined <statement>s. If an <invalid> statement or a <statement> not permitted by the syntax of a <statement unit> is encountered, then an error is reported, all input following the offending statement is flushed, and the statement itself is throw away. Additional lines may then continue to be entered.

When the translator has assembled enough <statement>s to form a single, syntactically complete <statement unit>, the <statement unit> is passed to the processor for execution.

If execution completes normally, control returns to the translator to read another <statement unit>.

Execution

An <active unit> is executed in a manner dependent on the form of the <active unit>:

Case 1. The <active unit> is a <statement unit>: Execute the <statement> or <group> that comprises the

<statement unit>.

Case 2. The <active unit> is a <program unit>: Activate the block denoted by the contained <procedure> at the entrypoint that was invoked, causing execution to commence.

The execution of an <active unit> moves from <statement> to <statement> along a path called the flow of control. The interpretation given to a <statement> is subject to the environment of execution as defined in the immediately containing <block>.

The rules for executing an individual <statement>s are given in the section on the Syntax and Semantics of Statements.

THE ENVIROMENT OF EXECUTION

Each <block> has associated with it a definition list, a resolution rule list, and an active handler list which together define the environment of execution.

The definition list is a list of <identifier>s defined in the block and the values or objects to which they are bound.

The resolution rule list specifies rules denoting which (<block>s') definition lists are to be searched when resolving an <identifier> reference.

The active handler list specifies a list of conditions for which handlers (<on unit>s) have been established in the <block>.

In addition to those defined for <block>s, there is a global definition list maintained for the purpose of having definitions that last from process to process, and a definition list giving the builtin functions and psuedo-variables defined by the language. (See the section on Builtins.)

The execution of a <statement unit> (which is not contained in any <block>) is influenced by environment lists specially maintained in the invocation of the command interpreter directing the processing of the <statement unit>.

BLOCK ACTIVATION

A <block> is activated when an entry to a <procecure> is invoked or an <on unit> is invoked to handle a condition. To activate a <block>, perform the follow operations:

1. Initialize the resolution rule list by executing the <Environment statement> immediately contained within the

<block> if one exists. Otherwise, execute the default <Environment statement> as described in the description of that statement.

2. Initialize the active handler list to null.

3. Initialize the definition list to null, and process any contextually derived definitions to be made in the <block>. A definition may not override a previous definition of the same <identifier>.

   a. For each <label> that is immediately contained in the <block>, create a definition for the <identifier> specified in the <label>, and bind it to a label value designating the following <executable unit>.

   b. For each <procedure> that is immediately contained in the <block>, create definitions for the <identifier>s naming the <procedure statement> and any <Entry statement>s immediately contained in the <procedure>, and bind those <identifier>s to entry values designating the corresponding entrypoints.

4. Execute each non-"Local" <scope statement> and <Synonym statement> immediately contained in the <block>.

5. Create a definition for each <identifier> that appears as a parameter in a <procedure statement> or a <Parameter statement>, and process the <parameter block> or (implied) <parameter list> designated for the entrypoint for the block.

6. Execute each "Local" <scope statement> immediately contained in the block.

7. Excluding the <descriptive> and <declarative> statements just processed, the body of the <block> forms a list of <executable unit>s. Execute these <executable unit>s according to the flow of control beginning with the first such unit that follows the point at which the <block> is to be entered.

The <block> that has been activated most recently is called the current block. The <block> that invoked the current block is called the calling block. The <block> that immediately contains the current block is called the parent block.

THE FLOW OF CONTROL WITHIN A BLOCK

The <statement>s or <group>s that comprise a list of <executable unit>s are executed in the order in which they appear, except as the flow of control is influence by the execution of individual <statement>s. Upon the completion of the list, control returns to the point at which the <block> was invoked.

The execution of a <Goto statement> can cause execution to move to a <executable unit> other than the next one in sequence. In such a case, execution continues as if that <executable unit> (the target of the goto) were reached normally from its preceding statement.

When an <if group> is encountered during execution, an <executable unit> contained in the <then part> or <else part> may be selected for execution. If such a case occurs, that <executable unit> is executed normally. Execution then proceeds normally to the next unit.

When a <do group> is encountered during execution, the contained <statement list> is executed as a list of <executable units> subject to the control of the <Do statement>. When execution of the <do group> is complete, execution continues with the next unit.

EVALUATION OF EXPRESSIONS

The Data of ECL

The data that is manipulated by the language takes on two forms: the simple constant values produced as the result of evaluating an expression, and data objects which may be assigned any desired value and have certain properties relating to the way in which their values are accessed.

Data Types

Each value has associated with it one of fifteen data types that determine how it is stored internally and what operations may be performed upon it. The data types that are supported are:

1. integer - represents positive and negative whole numbers and is stored internally as fixed binary(35).

2. real - represents arithmetic values with fractional parts, and is stored internally as float decimal(14).

3. logical - represents a simple truth value and is stored internally as bit(1) aligned.

4. string - represents character strings or bit strings (with length greater than one) and is stored internally as character(256) varying.

5. literal - represents strings that have a special meaning in the language or a special meaning to user defined statements (commands) such as keywords, operators, and punctuation. Literal values are stored internally as character(32) varying.

6. address - represents the address of a storage location and is stored internally as a pointer.

7. date - represents a Multics standard clock reading (microseconds since January 1, 1900) and is stored internally as fixed binary(71).

8. pathname - represents an absolute pathname of some directory entry and is stored internally as character(168).

9. branch - represents a directory entry and is stored internally as a structure containing relevant information about the entry.

10. iocb - represents an I/O switch and is stored internally as a pointer to an I/O control block.

11. refname - represents a reference name and is stored internally as character(32).

12. label - represents a location in a program and the environment of the invocation of the program. It is stored internally as a label value.

13. entry - represents an entry into a subroutine and the environment of the block in which the subroutine is defined. An entry value may represent both command or noncommand subroutines. The former are procedures defined by or written in the language. The latter include all procedures written in other languages and still callable from within the language. An entry value may represent three types of subroutines:

    a. A procedure is a subroutine which may be invoked by a simple call and which executes a certain sequence of operations.

    b. A function is invoked to compute and return a value.

    c. A psuedo-variable may be invoked to return a value or used as the target of an assignment to receive a supplied value. Psuedo-variables are useful to model values in the command environment which require subroutine calls to alter, e.g. the working directory.

An entry value is stored internally as an entry value, with additional designators indicating whether it is a command/noncommand procedure and whether it is a procedure, function or psuedo-variable.

14. undefined - represents the absence of a value of any other type. It is the value associated with variable objects bound to newly defined identifiers that have no other initial value specified.

15. list - is an aggregate type that is an ordered sequence of values which may be accessed as a group or individually. It may be used to represent arrays, structures, stacks, or abstract data types. There is one element corresponding to each positive integer. An element that has never acquired a value in any manner has the type undefined. A list has one attribute, its length, which gives the largest index for an element with a nonundefined value. If then there are no such elements, then the length is zero.

Object_Types

1. Simple variable objects which have an associated value that may be changed by assignment.

2. An external data object describes an external symbol -- for example, an error_table_ code -- giving its name and a fixed type. Evaluating an identifier bound to such an object will extract the value from the external location; assigning to the value of the identifier will alter the value of the external location. It may have any data type except entry.

3. A psuedo-variable reference describes a particular use of a psuedo-variable giving the arguments with which the procedure is to be invoked. It is created when a psuedo-variable used as the target of an assignment or passed by reference to a subroutine.

4. A list cross-section describes a particular sublist of a list value. It designates the list, the object having the list as a value, the starting element of the sublist and the length of the sublist. It is created to describe a list cross-section ocurring as the target of an assigment or being passed by reference to a subroutine.

Structure of Expressions

There are two syntactic types of expressions: basic expressions which either are single tokens or are delimited by parentheses or braces and which are used primarily as single arguments, and expressions which involve several tokens and must appear delimited by some keyword pharse or punctuation. Expressions represent some computation to be performed.

<basic expression> ::=
<identifier> | (<expression>) | <constant> | <list>

<constant> ::= <string> | <literal>

<string> ::= <quoted string> | <unquoted string>

<literal> ::= <option name>

<list> ::= { [<basic expression>...] }

<expression> ::= <infix> | <prefix> | <combination>

<infix> ::= <expression> <infix-op> <expression>

<prefix> ::= <prefix-op><expression>

The result of evaluating a basic expression is the result of evaluating the contained <identifier>, <constant>, <list> or <expression>. The result of evaluating an <expression> is the result of evaluating the infix or prefix operation or <combination> it represents. The methods of evaluating these inferior constructs are described below.

Constants

There are two types of constant values that may appear as or in an expression.

1. A <string> represents to a data value of type string.

2. A <literal> represents a literal data value corresponding to the designated <option name>.

Note that the class of unquoted strings also includes what would normally be considered as numbers -- 123, 12.3, 12e3 and so forth; these are considered strings until the context of their use forces conversion to arithmetic (integer or real) values.

Identifiers

Identifiers are the names of objects or constant values. They are a subset of unquoted strings. They must begin will an uppercase alphabetic character and contain only alphabetic characters and digits and the characters "_", "$", and "%".

DEFINITION OF IDENTIFIERS

The creation of a definition for an identifier appends the <identifier> to a definition list and binds the <identifier> to an object or value. All <identifier>s must be defined before they are used. There are five means of definition:

1. Definition of identifiers representing simple variable objects. (See the scope statement.)

2. Definition of identifiers representing external data objects. (See the Synonym statement.)

3. Definition of an identifier representing an entry constant. (See <procedure> and the <Entry statement>.)

4. Definition of identifiers representing constant label values.

5. Definition of the pararameters to a subroutine. An identifier that is a parameter may be bound to either a constant value or an object of any type.

RESOLUTION OF IDENTIFIER DEFINITIONS

An <identifier> definition is resolved by finding the value or object to which it is bound. To resolve a definition, perform the following procedure:

1. Search the definition list for the current block for a definition of the <identifier>. If found, then return the object or value to which the <identifier> is bound.

2. Otherwise, the <identifier> definition is to be resolved subject to the the list of <resolution rule>s established by the (implied) Environment statement for the current <block>: Apply the rules specified in the designated order. The rules are interpreted as follows:

   a. The <u>Previous</u> rule specifies that the definition list in the immediately previous activation of the containing <block> is to be searched. If no such activation exists, then this rule is skipped.

b. The <u>Parent</u> rule specifies that the search is to continue in the parent of the current block -- the <block> immediately containing the current <block>, following the rules established in the parent. If there is no parent block, then this rule is skipped.

c. The <u>Caller</u> rule specifies that the search is to continue in the block that invoked this block -- the block that invoked a subroutine, entered a begin block, or signalled the condition invoking an on unit -- following the rules established in the caller. If there is no caller, then this rule is skipped.

d. The <u>Global</u> rule specifies that the global definition list is to be searched.

e. The <u>Builtin</u> rule specifies that the list of builtin functions and psuedo-variables is to be searched.

f. The <u>External</u> rule specifies that the user search rules are to be used to attempt to find an external symbol with the name of the identifier. Resolution of an identifier by this rule causes an entry value designating the external entry point found to be returned.

3. The definition list selected by the application of the rules above, is searched for an non-transparent definition of the <identifier>. If one is found, then the object or value to which the <identifier> is bound is returned. If the search fails, the next rule is applied until the list is exhausted.

EVALUATION OF AN IDENTIFIER

When an <identifier> is used as part of an expression, it is evaluated to find the value that it currently represents. To evaluate an <identifier>, first, resolve the <identifier>'s definition. If it is bound to a constant value, then return that value as the result of evaluating the <identifier>. If it is bound to an object of some sort, then evaluate that object and return the result.

<u>Combinations</u>

A <combination> is used to represent parenthesized expressions, function invocations, list cross-sections, and psuedo-variable invocations.

```
<combination> ::= <constant>|<list>|
                                (<expression>)|<reference>
```

Evaluation depends on the nature of the <combination>.

Case 1. The <combination> is a <constant>, <list>, or parenthesized <expression>: Evaluate the construct, and return the result as the result of evaluating the <combination>.

Case 2. The <combination> is a <reference>: Evaluate the <reference>. If the result is a constant value, then return that value. Otherwise, evaluate the object, and return the result.

References

A <reference> represents a data object or value. It is used to describe the target of an assignment, the arguments to a subroutine, and the operand of an expression.

<reference> ::= <simple reference> | <complex reference>

<simple reference> ::= <identifier>

<complex reference> ::= <element><argument list>

<element> ::=
<constant>|<identifier>|(<expression>)|[<reference>]

To evaluate a <reference>, select the applicable case, and perform the indicated operations.

Case 1. The <reference> is a <simple reference>: Evaluate the <identifier> specified, and return the value or object to which it is bound.

Case 2. The <reference> is a <complex reference>: Evaluate the component <basic expression>s and <reference>s of the <argument list> in an unspecified order. Evaluate the <element>, and select the applicable case:

a. If the evaluated <element> is a list value, or is an object representing a list value or cross-section, then the evaluated <argument list> must consist of one or two values which must be convertible to integer values. Let i be the converted value of the first expression. If there is a second, let n assume its value; otherwise, let n be 1. The result is a list cross-section of the object or value representing the n elements beginning with the ith element.

b. If the evaluated <element> is a value that converts to an entry value representing a function, then derive and process the arguments as for a procedure call. Invoke the function with these arguments and return the result.

c. If the evaluated <element> is a value that converts to an entry value representing a psuedo-variable, then derive the arguments to be passed to the psuedo-variable when it is invoked as for a procedure called. The result is a psuedo-variable reference formed by associating the arguments with the psuedo-variable.

d. All other cases are in error.

Object Evaluation

The evaluation of an object yields the value associated with that object:

1. If the object is a variable object, then the value associated with the variable object is returnec as the result.

2. If the object is an external data object, then the result is a value with the value extracted from the external location.

3. If the object is a psuedo-variable reference, then the psuedo-variable designated is invoked with the associated arguments, processed as for a procedure call. The value returned by the psuedo-variable is the result.

4. If the object is a list cross_section, then the object representing the list is evaluated, and a list consisting of the designated elements returned.

Lists

The <list> construct, <basic expression>s enclosed in braces, evaluates into a list value. The elements are formed by evaluating the expressions, and forming the resulting values into a sequence ordered left to right.

Operators

The language supports most of the standard infix (two operand) and prefix (one operand) operators, as well as a few special ones. Evaluation of an infix or prefix operator causes the operands to be evaluated, the indicated operation to be performed, and the result returned as the value of the operation. There are five types of operators: arithmetic, logical, comparision, string and list. They maintain the normal (i.e. PL/I) operator precedence.

Most of the operators normally work on scalar data values. If the one operand of a prefix operator is a list, then the result is a list of the variable objects having the values derived from applying the operator to each element of the list individually. For infix operators, if both operands are lists (of the same length), then the result is a list of the values resulting from a pairwise application of the operator to the elements of the two lists. If one operand is a scalar, and the other is a list, the scalar is promoted to a list of the appropriate length.

Except as otherwise noted, it is an error for any of the operands to have an Undefined value. the program is in error.

ARITHMETIC OPERATORS

These perform the standard arithmetic operations between their operands. The operands are expected to be either integer or real. If they are not, they are converted to one of these types, as appropriate (see conversions), before the operation is attempted. The result will be integer if there is enough precision to hold the result, otherwise the result will be real. There are five arithmetic operators:

+	addition
-	subtraction (infix), negation (prefix)
*	multipication
/	division
**	exponetiation

LOGICAL OPERATORS

There are two logical infix operators and ("&") and or ("|"), and one logical prefix operator not ("¬"). These operators expect their operands to be of type logical, and as above, operands not of this type will be converted. The result is a logical data value.

COMPARISION OPERATORS

These operations return a logical value indicating whether or not the specified comparision was successful. The comparisions fall into two categories: value comparision, and type comparison.

Value Comparision

There are six infix operators which may be used to compare the values of types integer, real, logical, string, and literal. They are the standard operators:

=	is equal to
^=	is not equal to
<	is less than
<=	is not greater than
>=	is not less than
>	is greater than

These operators expect there operands to be of the same type. If not they are converted according to the table below. Comparisions of values of type integer and real are done arithmetically. Comparisions of logical are performed as if the values were the integers, 0 or 1. Comparisions of string and literal values are done according to the ASCII collating sequence.

	integer	real	logical	string	literal
integer	integer	real	integer	integer	string
real	real	real	real	real	string
logical	integer	real	logical	logical	string
string	integer	real	logical	string	string
literal	string	string	string	string	literal

These six operators may also be used to compare values of type date. If one operand is not a date value, it must be a string convertible to a date value. The comparision is performed arithmetically on the internal fixed binary(71) form.

Only the operators "=" and "^=" may be used to compare values of other data types. Addresses compare equal if they specify the same location. Entries and labels compare equal if thet describe the same location and generation of storage. Iocbs compare equal if they identify the same I/O switch (even if syn_'ed). Branches compare equal if the segments have the same unique id's. Pathnames compare equal if they describe the same directory entry. Comparisions are made between values of the same type. If one of the values is string, and the other address, entry, branch, pathname, refname, or iocb, then the string is converted to the other type. If both values are of the types refname, pathname, address or branch, then conversion to address or branch (if one of the two is a branch) is attempted before the comparision. Otherwise, the program is in error. It compares equal to only another undefined value.

Type Comparision

The operator, "==", is an infix comparision operator returning the value True if the two operands are of the same type. It has the same precedence as "=".

STRING CONCATENATION

The operator, "||", is used to concatenate two strings together. It expects both operands to be of type string. If they are not, the offenders are converted before the operation is performed. The result is a string value.

LIST CONCATENATION

The infix operator, "!", is used to join two lists together in the same manner as two strings are joined by concatenation. It expects both operands to be of type list; if not, the scalar operands are promoted to one element list before the operation is performed. It has the same precedence as "||".

Conversions

Conversions may be requested explicitly by use of conversion functions, or implicitly by context. The conversions given in the table below may be performed among scalar data types.

An equals sign indicates conversion of a value to the same type, in which case the value is simply copied. An asterisk indicates a conversion that may be performed under transitive closure and which is likely to have meaning. These multistep conversions will be performed automatically. (An attempt can be made to convert most types to most any other type with a string value as an intermediate step, but such attempts will generally result in conversion errors.) Those marked with numbers indicate one step conversions and are described below.

Data Type Conversions

FROM \ TO	integer	real	logical	string	literal	address	date	pathname	branch	iocb	refname	entry	label
integer	=	1	3	8									
real	2	=	*	8									
logical	4	*	=	9									
string	5	6	7	=	11	12	13	14	*	18	20	27	
literal				10	=								
address				12		=		22	*		24		
date				13			=						
pathname				15		23		=	16		*		
branch				*		*		17	=		*		
iocb				19						=			
refname				21		25		*	*		=		
entry				26								=	
label													=

1. integer to real - the integer value is converted to a real value according to the rules of PL/I for their internal forms.

2. real to integer - the integer part of the real value is taken. For example, 2.34 becomes 2; -2.34 becomes -2.

3. integer to logical - if the integer value is nonzero, the result is True; otherwise, it is False. Real values are converted to integer in conversion to logical.

4. logical to integer - True becomes the value 1; False, zero. Logical to real involves a conversion through integer.

5. string to integer - if the string represents an integer in the range -2**35 to 2**35-1, the result is that integer. If the string represents a real value in that range, the integer part of that value is the result. Otherwise, the program is in error.

6. string to real - if the string represents a number in the range -10e128 to +10e127, the result is that number. Otherwise, the program is in error. Whenever a string is to converted to an arithmetic type (as for example, when it is the operand of an arithmetic operator) a string to integer conversion will be performed if the number is an integer in the appropriate range; otherwise, a string to real conversion will be attempted.

7. string to logical - if the string is "1", then the result is true; if the string is "0", then the result is false. Otherwise the program is in error.

8. integer or real to string - the result is the character string which most compactly represents the number. There will be no leading blanks, and a sign will appear only for negative values. For values with magnitudge greater than 10**14, exponential form will be used.

9. logical to string - if the source value is true, the result is "1"; otherwise, the result is "0".

10. literal to string - the character string representing the literal is simply copied.

11. string to literal - the character string representing the string value is simply copied. If, however, there are more than 32 significant characters in the string, the program is in error.

12. address to string, string to address - the ioa_ format for a pointer is used to represent the address value as a string. If a string to address conversion fails, a string to pathname to address conversion will be attempted before an error is reported.

13. date to string, string to date - the convert_date_to_binary_ format string is used to represent the string equivalent of a date.

14. string to pathname - a (possibly) relative pathname is expanded to an absolute pathname. The entry portion of the pathname may be star laden.

15. pathname to string - the character string representing the pathname is simply copied.

16. pathname to branch - the pathname is copied and a check is made to verify that the entry specifed by the pathname actually exists. (Star laden entry names will result in an error.) Other conversions to branch are done via pathname.

17. branch to pathname - the pathanme of the directory entry is copied. Conversions from branch values are performed with pathname as the intermediate type.

18. string to iocb - the I/O control block for the I/O switch whose name is given in the string is found.

19. iocb to string - the name of the I/O switch associated with the IOCB is the result.

20. string to refname - the string is copied, and a check is made to verify that it is indeed a reference name on some segment.

21. refname to string - the character string representing the refanme value is copied.

22. address to pathname - the result is the pathname of the segment designated by the address value.

23. pathname to address - the segment specified by the pathname is initiated.

24. address to refname - the first nonnull reference name on the segment specified by the address value is used.

25. refname to address - a pointer to the segment whose reference name is given is found.

26. entry to string - if the entry represents a command procedure, the result is the identifier associated with that entry. If the entry represents an external procedure, then the result is the name of the entry in the form segname[$offsetname].

27. string to entry - first, a check is made to see if there is a procedure, function or psuedo-variable with the name given by the string. If so, the result is an object of the appropriate type with the address and environment of the routine specified. Second, a search for an external procedure is performed according to the algorithm of find_command_, and if found becomes the entry value. It will be a procedure or function depending on the status of the function bit in the entry parameter descriptor list (if one is present); otherwise, it will be designated a procedure.

Promotion

The promotion of a scalar to a list causes a list of the appropriate length to be constructed from copies of the value fo the scalar.

THE SYNTAX AND SEMANTICS OF STATEMENTS

Independent Statements

THE Call STATEMENT

<basic expression> [<argument list>]

<argument list> ::=
{<basic expression>|[<reference>]}...

The leading <basic expression> may not be a simple <identifier>. This restriction is made to permit leading <identifier>s to identify other statements.

This statement causes a designated procedure to be invoked with the argument list supplied. To execute a <Call statement>: Evaluate the leading <basic expression>, and convert the result to an entry value. This value must represent a procedure as opposed to a function or psuedo-variable. If there is no <argument list>, then invoke the procedure without arguments; otherwise, evaluate each component of the <argument list>, and derive the arguments to be passed to the called procedure in the following manner:

1. A <basic expression> that evaluates to a scalar (nonlist) value, produces a single argument, the value of the <basic expression>.

2. A <basic expression> that evaluates to a list value produces a number of arguments that is equal to the length of the list (including zero). The arguments (ordered left to right) are the values that are the elements of the list.

3. A <reference> produces a single argument which is the object that results from evaluation of the <reference>.

The arguments have the same left to right ordering as their generating syntactic forms. Prior to being passed to the procedure, the arguments are processed in a manner dependent on the type of the procedure:

1. If the procedure is a command procedure, then no further processing is required. The arguments are passed as is, to be processed according to the parameter specification

of the called procedure. The procedure may assign values to those arguments which are objects (i.e. result form the evaluation of a <reference>).

2. If the procedure is an noncommand procedure with entry parameter descriptors, then the arguments are first evaluated and converted to the type expected by the procedure. Scalars convert to their corresponding PL/I types. Lists may convert to either one dimensional arrays or structures (with contained lists matching substructures). On return, the argument values corresponding to <reference>s, but possibly modifed by the called procedure, are reassigned to the original <reference>.

3. If the procedure is an noncommand procedure without entry parameter descriptors, then the arguments are passed according to their corresponding PL/I data types. Lists are passed as uni-dimensional arrays if their elements are convertible to a common type. (The common type is determined as for the comparison of two different data types.) On return, the argument values corresponding to <reference>s, but possibly modifed by the called procedure, are reassigned to the original <reference>.

THE Let STATEMENT

Let <reference> = <expression>

Execution of a <Let statement> causes the source <expression> to be assigned to the target <reference> to alter the value of the object that the latter represents.

To execute a <Let statement>, evaluate the source <expression> and the target <reference>, and assign the value of the source expression to the evaluated target. The target of an assignment operation may not be a constant value. Otherwise, assignement is performed by selecting the applicable case on the basis of the type of the target.

Case 1. The target is a variable object: associate the object with the value of the source expression.

Case 2. The target is an external data object: Convert the source value to the type of the external object, and copy the result into the external location.

Case 3. The target is a psuedo-variable reference: Invoke the psuedo-variable with the arguments specified, and supply it with the assigned value.

Case 4. The target is a list cross-section: Evaluate the object representing the list value. Insert the element(s) of the source value into the resulting list value in place of the designated element(s) extending or contracting the list as necessary. Assign the modified list to original object.

THE Exit STATEMENT

Exit

An <Exit statement> may not appear immediately contained in the body of a <procedure>.

The execution of an <Exit statement> is dependent on the context in which it appears:

Case 1. It appears within a <group> as an <executable unit>: Terminate the execution of the <group>.

Case 2. It appears in the body of an <on unit> as an <executable unit>. Return from the <on unit> to the point at which the condition was signaled.

Case 3. It appears in a the body of a <for unit> as an <executable unit>. Return from the <for unit> to the <Perform statement> invoking the <for unit>.

THE Continue STATEMENT

Continue

A <Continue statement> may only appear as an <executable unit> in a <group> headed by an <iterative do>, a <do while>, or a <do list>.

To execute a <Continue statement>, terminate the execution of the list of <executable unit>s in the <group>'s <statement list>, and continue with the next step in the execution of the <group>.

THE Goto STATEMENT

Goto <identifier>

The <identifier> must evaluate to a label value.

To execute a <Goto statement>, evaluate the <identifier>. It must yield a label value. Move control to the <executable unit> designated by the label value bound to the <identifier>.

This operation involves a local goto if the statement is in the current block. If the statement is in some other block, this involves a nonlocal goto which causes all intervening active blocks to be deactivated and the condition cleanup to be signalled in each such block before deactivation.

THE Interpret Statement

Interpret

The <Interpret statement> allows the construction of a command interpreter using the mechanisms of the command language interpreter for input and translation.

To execute an <Interpret statement>, read and translate one <statement unit> (as from command level), and execute in it as if it were contained in the <block> containing the <interpret statement> itself.

THE Perform STATEMENT

Perform

The <Perform statement> is used to invoke a <for unit body> comprising the second part of a compound command.

To execute a <Perform statement> simply invoke the <for unit body> contained in the <for unit> that invoked the <procedure> containing the statement. If no such <for unit> exists, the execution of the <Perform statement> will have no effect.

THE Signal STATEMENT

Signal <condition> [<argument list>]

The <Signal statement> is used to invoke the most recent handler for a specified condition. It may cause a handler established by the standard Multics condition mechanism to be invoked. To execute a <Signal statement>, perform the following procedure: Evaluate the <condition>, and convert its value to a string value. This yields the name of the condition. Begin a search for a handler for the condition in the current block, and continue, on failure, with its callers. Scan the active handler list of the block being searched first for a handler for the condition specified, and then for a handler for the condition "any_other". If the search yields a handler, then process the <argument list> as for a procedure call, and invoke the handler. If no handler is found, the program is in error.

Note also that as in a <procedure>, while executing in an <on unit>, the values of the arguments passed by reference may be changed to communicate information back to the calling procedure (the procedure signalling the condition).

THE Revert STATEMENT

Revert <condition>

To execute a <Revert statement>, evaluate the <condition> expression and convert the result to a string value giving the name of the condition to revert. Search the active handler list in the current block for a handler for the condition so named. If one is found, remove it from the list.

THE Return STATEMENT

Return [<expression>]

A <Return statement> can only appear immediately contained within body of a <procedure> or a <data unit>. The optional return <expression> may only appear in the body of a function or psuedo-variable; the return <expression> must appear when in the <data unit>.

In the context of a <procedure> body, execution of this statement causes the subroutine in which it is contained to return to the point at which it was invoked. If the optional <expression> is present, it is returned as the value of the function or psuedo-variable. (If a psuedo-variable was invoked to receive a value instead of return a value, then the program is in error if the <expression> is given.)

In the context of a <data unit>, execution of this statement causes the <expression> to be evaluated, converted to a character string, and returned as the next value of the <data unit>.

THE Resignal STATEMENT

Resignal [<argument list>]

A <Resignal statement> may only appear immediately contained in the body of an <on unit>.

Execution of this statement causes the current <on unit> to be exited. The search for an active handler for the condition, as performed by a <signal statement>, is continued, and the next most recent handler invoked. If an <argument list> is specified, it is processed as for a procedure call, and the next handler is invoked with that group of arguments. If no <argument list> is specified, the same (but possibly altered) argument list as the current handler was invoked with is used.

THE NULL STATEMENT

<null statement> ::= no tokens

Execution of a <null statement> causes no action to be performed and has no effect on the program. Control passes normally to the next <executable unit>. The purpose of the <null statement> is to provide, for example, a convenient way to specify a <then part> that performs no action.

Dependent Statements

THE If STATEMENT

<If statement> ::= If <basic expression>

The <If statement> controls the internal execution of an <if group>. That is, it selects for execution either the <then part> or optional <else part>.

To execute an <If group>, evaluate the <basic expression> appearing in the <if statement>, and convert its value to a logical value. If the result is true, execute the <executable unit> contained in the <then part>; otherwise, if a <else part> is given, execute the <executable unit> contained in the <else part>.

THE Do STATEMENT

```
<do statement> ::= <simple do>|<iterative do>|
                        <do while>|<do case>|<do list>

    <simple do> ::= Do

    <iterative do> ::=
            Do <do index> = <initial> Repeat <next>
                                      [<while precicate>]

        <do index> ::= <reference>
        <initial> ::= <expression>
        <next> ::= <basic expression>
        <while predicate> ::= While <basic expression>

    <do while> ::= Do <while predicate>

    <do case> ::= Do Case <case selector>
```

<case selector> ::= <basic expression>

<do list> ::= Do <list spec>[,<list spec>]...
[<while predicate>]

<list spec> ::= <do index> From <list value>
<list value> ::= <basic expression>

The <do statement> denotes the beginning of a <group> and controls the execution of the <executable unit>s contained in the <group>'s <statement list>.

A <do statement> is never itself actually executed. Rather, when the control encounters the <do group> as an <executable unit>, the applicable case is selected on the basis of the <do statement> and the indicated operations performed.

Case 1. The <do statement> is a <simple do>:

Execute the list of <executable unit>s once, then terminate the execution of the <group>.

Case 2. The <do statement> is an <iterative do>:

a. Evaluate the <reference> in the <do index>; let the result be R. Evaluate the <initial> expression, and let its result be V.

b. Assign V to R.

c. If a <while predicate> is given, evaluate the <basic expression>, and convert the result to a logical value. If this value is false, the execution of the <group> is complete.

d. Execute the list of <executable unit>s, and when finished, continue to step e.

e. Evaluate the <next> expression, and let the result be V. Continue with step b.

Case 3. The <do statement> is a <do while>:

a. Evaluate the <basic expression> given in the <while predicate>, and convert the result to a logical value. If the value is false, then the execution of the <group> is complete.

b. Execute the list of <executable unit>s, and continue with step a.

Case 4. The <do statement> is a <do case>:

a. Evaluate the <case selector> expression, and convert the result to an integer value. Let I be this value. The program is inerror if I <= 0 or if I is greater than the number of <executable unit>s in the <statement list>.

b. Execute the Ith <executable unit> in the list.

Case 5. The <do statement> is a <do list>:

a. Evaluate each <reference> specified as a <do index>; let the results be R*1*, ..., R*n* where *n* is the number of <list spec>s. Evaluate each <list value>. If the result is not a list, promote the scalar value to a one element list. Let the list values be L*1*, ..., L*n*. Let *k* be 1; let *l* be the length of L*1*. The program is in error if *l* does not also equal the length of all the other L*i*.

b. Assign to each R*i* the *k*th element of L*i*.

c. If a <while predicate> is given, evaluate the <basic expression>, and convert the result to a logical value. If the value is false, then execution of the group is complete.

d. Execute the list of <executable unit>s and when finished, continue with step e.

e. Let *k* be the value of *k* + 1. If *k* > *l*, then execution of the group is complete. Otherwise, continue with step b.

THE For STATEMENT

<For statement> ::= For <Call compound>

The <For statement> is used to construct compound commands. It denotes the beginning of a group of <statement>s that are subject to the control of the command specified in the <For statement> itself.

A <For statement> is never actually executed. Rather when control reaches the <for unit>, the <Call statement> specified is executed. Execution of this command may cause the <for unit body> to be invoked by execution of a <Perform statement>.

After the <for unit> has been invoked, control is returned to the point at which it was invoked, that is, the <Perform statement>, upon completion of all <executable unit>s in the body of the <for unit> or upon execution of an <Exit statement>. The

<for unit> may be invoked zero or more times by the specified command.

The <for unit> may be viewed as a single entry procedure with no arguments that is passed to the command specified. However, its environment is normally the same as an internal procedure defined __within__ the command. The <Environment statement> may, of course, change this.

THE Data STATEMENT

<Data statement> ::= Data <reference>

The <Data statement> defines a block of statements that generate lines of input to be read.

The <Data statement> is not itself directly executable. Rather, when control encounters the <data unit> as an <executable unit>, an IOCB value for a switch controlling this input stream is assigned to the <reference>. When subsequent attempts are made to read from this switch, the <data unit> is invoked to return a value that is to be convertec into a character string and "read" as input. For the first such invocation, control begins with the first <executable unit> in the <data unit>, and ends when a <Return statement> is executed; for all subsequent invocations, control resumes at the point following the <Return statement> previously executed, and again terminates when the next <Return statement> is executed.

THE On STATEMENT

On <condition> [<parameter list>]

The <On statement> denotes the beginning of an <on unit>, a handler for an abnormal condition, which may be viewed as a single entry procedure that is invoked when the condition is signalled. The <On statement> further defines the parameters with which the handler is to be invoked.

The <On statement> is not itself directly executable. Rather, when control encounters the entire <on unit> as an <executable unit>, the <on unit> is establised as a handler for the specified condition by performing the following procedure: Evaluate the <condition> expression and convert the result to a string value. This is the name of the condition. Add the <on unit> to the active handler list in the current block as a handler for the condition replacing any handlers for the condition previously established.

The condition may be signalled, and the <on unit> invoked, by the execution of a <Signal statement> (see above) or by the standard Multics signalling mechanism.

The parameters are specified for an <on unit> in the same manner as for a <procedure>. Either a <parameter list>, a <parameter block>, or the Argument builtin may be used.

After an <on unit> as been invoked, control is returned to the point at which the condition was signalled upon completion of the execution of all <executable unit>s in the body of the <on unit> or upon execution of an <Exit statement>.

THE PROCEDURE STATEMENT

```
<procedure statement> ::=
          <type descriptor><identifier>[<parameter list>]

     <type descriptor> ::= Procedure|Function|Variable
```

The <procedure statement> denotes the beginning of a <procedure>, a subroutine block, and designates the type of the subroutine by the <type descriptor>. Moreover, it defines an entrypoint to the <procedure> and may include a description of the <parameter list>.

A <procedure statement> is not itself directly executable. The appearance of a <procedure statement> in a <block> causes the <identifier> specified to be defined and bound to an entry value desginating the corresponding location when the <block> is entered. The appearance of a <procedure statement> in a <procedure> that is a <program unit> causes a corresponding external entry to be genrated for the the object segment into which it is compiled. Parameter Specification

Parameter Specification

The parameters to a procedure, function, or psuedo-variable are specified in one of two ways: by providing a <parameter list> in the procedure header, or by giving a <parameter block> defining the syntax and semantics of the arguments expected by the procedure.

If a <parameter block> appears within the <body> of a procedure, it is taken to apply to all entries to that procedure unless there is a <parameter block> which appears immediately following each entry, in which case each such block applies to its corresponding entry. If there are no <parameter block>s within a procedure, then each entry is considered to have an (implied) <parameter list>. All other cases are in error.

There is one final mechanism for referencing the arguments to a procecure: the builtin list Argument, which corresponds to the argument list with which the procedure was invoked.

Parameter lists

A parameter list specifies a list of <reference>s which will assume the identity of the values passed to the entry that the parameter list applies to:

<parameter list> ::= {<value parm>|<reference parm>}
<value parm> ::= <identifier>
<reference parm> ::= [<identifier>]

when the entry to which the <parameter list> applies is invoked, one of two operations will be performed for each parameter-argument pair:

Case 1. The parameter is a <value parm>: The corresponding argument must be a simple value. Bind the <identifier> to the value given.

Case 2. The parameter is a <reference parm>: The argument must be some sort of object. Bind the <identifier> to the object given.

If an entry is invoked with less arguments than there are parameters specified for the entry, then the program is in error. If an entry is invoked with more arguments than there are parameters specified, then it will be expected that the remaining arguments are to be referenced by the Argument builtin psuedo-variable, and no error will be reported.

THE Entry STATEMENT

Entry <identifier> [<parameter list>]

The <Entry statement> defines an alterante entrypoint for a <procedure>. The type of entry value that it designates is given by the <type descriptor> in the <procedure statement> beginning the <procedure>. It may also describe the parameters to the entry.

An <entry statement> is never directly executed itself. The appearance of an <Entry statement> in a <procedure> causes an additional <identifier> to be defined as entries to the subroutine in the same manner as the <identifier> appearing the <procedure statement>.

THE End STATEMENT

End

The purpose of the <End statement> is to syntactically close the constructs: <procedure>, <on unit>, <do group>, <parameter block>, and <compound form>. It is never actually executed.

Declarative Statements

THE SCOPE STATEMENT

```
<scope><symbol spec>[,<symbol spec>]...

     <scope> ::= Local|Temp|Global|Parent|Caller|
                               Builtin|External|Previous
     <symbol spec> ::= <identifier>[<initialization>]
     <initialization> ::= = <expression>
```

An <initialization> option may not appear if the <scope> is other than Local, Temp, or Global.

The <scope statement> serves two purposes: to create definitions for local and global identifiers bound to variable objects, and to override the effect of an <Environment statement> for evaluating a single name. When making definitions, only a <scope statement> that is a <statement unit> may override previous declarations of the same <identifier> made in the <block>.

To execute a <scope statement>, select the applicable case and perform the indicated operations.

Case 1. The <scope> is "Local": Compute the initial value (as described below), let the result be V. Append a definition for the <identifier> to the definition list for the current block (subject to he restrictions concerning overriding a previous definition). Bind the <identifier> to a new variable object, and assign V to the object.

Case 2. The <scope> is "Temp": Compute the initial value, and let the result be V. Append a definition for the <identifier> to the definition list for the current block (subject to he restrictions concerning overriding a previous definition) with the notation that the definition is transparent (not to be found from other than the current block). Bind the <identifier> to a new variable object, and assign V to the object.

Case 3. The <scope> is "Global": Search the global definition list for a declaration for the <identifier>. If present, let the variable object to which it is bound be R. Otherwise, create a global definition for the <identifier>: Compute the initial value, and let the result be V. Append a definition to the global definition list. Bind this global instance of the <identifier> to a new variable object, also designated R. Assign V to R. In either case, add a definition for the <identifier> to the definition list for the current block (subject to the restrictions concering overriding a previous definition), and bind it to R.

Case 4. The <scope> is any other valid <scope>: Resolve the <identifier> by searching the definition list designated by the <scope> interpreted as a <resolution rule>; let the result be R. If no definition exists, then the program is in error. Otherwise, add a definition for the <identifier> to the definition list of the current block, and bind the <identifier> to R.

The initial value is determined in the following fashion: If an <initialization> is given, evaluate the contained <expression>; the result is the initial value. Otherwise, the initial value is undefined. Note that the initial value is calculated before the new definition is made. Therefore:

```
Local A = A
```

creates a local copy of the value given by the <identifier> "A".

THE Synonym STATEMENT

A synonym for an external object may be defined with the synonym statement:

```
Synonym <identifier> <external symbol> <type>

    <external symbol> ::= <basic expression>
    <type> ::= <basic expression>
```

To execute a <Synonym statement>, evaluate the <external symol> expression, and convert the result to a string value to give the name of the external symbol to be represented. Evaluate the <type> expression. If the value is Undefined, the program is in error. If the value is of type entry, then the external symbol is taken to designate a procedure entrypoint, and a definition for the <identifier> is created, and the <identifier> is bound to the corresponding entry value. If the value is of some other type, an external data object of that type is created to represent the external location, and a definition for the identifier bincreated, binding the <identifier> to the object so created.

THE Environment STATEMENT

The <Environment statement> may be used to control which definition lists are searched to resolve an reference to an <identifier>.

```
Environment [<resolution rule> [,[<resolution rule>]...]

      <resolution rule> ::= {Parent | Caller} |
                        Previous | Global | Builtin | External
```

Only one <Environment statement> may appear in the body of a <procedure> or <on unit>.

Execution of this statement establishes, in the <block> in which it is contained, a list of <resolution rule>s specifying what <block>s' definition lists are to be searched, in left to right order, when an identifier definition is to resolved. (See the discussion of the resolution of <identifier> definiticns.)

Default Environment Statements

If no <Environment statement> appears in a <procedure> that is a <program unit>, then by default, the following is supplied:

```
Environment Caller, Global, Builtin, External
```

Similarly, any other <procedure>, or a <data unit> or <on unit>, that does not contain an <Environment statement> has the default:

```
Environment Parent
```

A <for unit> is intended to be executed within the environment of its caller, and therefore has the default:

```
Environment Caller
```

The special "block" that defines the execution environment for <statement unit>s, initially has rules corresponding to the following statement:

```
Environment Global, Builtin, External
```

Descriptive Statements

PARAMETER BLOCKS

This facility allows the user to define the syntax and semantics of a new command by providing a means of describing the form and meaning of the arguments expected by a command procedure. The same facility is available for functions, psuedo-variables, and on units.

Syntax of the Paramter Block

```
<parameter block> ::=
   <Parameter statement> [<parm spec>...] <End statement>

   <Parameter statement> ::=
      Parameters <identifier> [, <identifier>]...

   <parm spec> ::=
      <basic form> | <compound form> | <construction>

   <basic form> ::=
      [<form prefix>]{<keyword spec>|<type spec>|<value spec>}
         [<success>][<default>|<error>]

      <keyword spec> ::= Keyword <literal> [Or <literal>]...

      <type spec> ::= <type name> <identifier>

      <value spec> ::= Value <identifier>

      <success> ::= <executable unit>

      <default> ::= Default <executable unit>

      <error> ::= Error <executable unit>

   <compound form> ::=
      <compound header>[<parmspec>...]<End statement>
         [<success>] [<default>|<error>]

      <compound header> ::=  [<form prefix>]
         <Group statement>|<Select statement>|<Multiple statement>

         <Group statement> ::= Group [<title string>]
         <Select statement> ::= Select [<title string>]
         <Multiple statement> ::= Multiple [<title string>]

            <title string> ::= <string>

   <construction> ::= <Form statement>[<from spec>]
```

<form statement> ::= Form {List|String} <identifier>

<from spec> ::= From <parm spec>

Meaning of Semantic Forms

The <Parameter statement> defines the <identifier>s which are to be the parameters to the contianing procedure. Their values are determined by the parameter specifications given in the body of the <parameter block> as described below.

The arguments supplied to an entry for which a <parameter block> has been specified must be simple values and are are scanned right to left (first to last) matching each to a form specifed in the parameter specification list. If a match cannot be found, or if the are too many or too few arguments an error is reported as described below.

The three basic forms are used to match a single argument. An argument that is a literal constant can only be used to match a Keyword form. This permits their use as unambigous delimiters of argument groups.

The Keyword form gives a list of one or more keywords (literal constants) of which one is expected to match the argument occuring in the implied location. The argument itself must also be a literal constant. For example:

Keyword -brief Or -bf

means that either the control argument -brief or its abbreviation must appear.

A type form requires the presence of an argument of a type convertible to the specified data type. The converted value is assigned to the <reference> specifed in the type specification. A <type name> may be one of the data types supported in the language: Integer, Real, Logical, String, Literal, Address, Entry, Label, Pathname, Branch, or Iocb.

The Value form merely requires the presence of an argument in a given location. The value of the argument is assigned to the <reference>.

The compound forms allow the specification of positional order for a list of forms, or of selection among one or more distinguishable forms which may appear in an unordered fashion.

The Group form defines an ordered list of one or more forms that must match arguments in the precise order given in the semantic block. The first form in a group may not be optional.

The Select form demands the appearance of one member of a list of forms. There must be an unambiguous way to distinguish between each of the forms in the list.

The Multiple form is similar, but requires the presence of one or more members of the list of forms. They may appear in any order, but one member of the list is permitted to be used only once. There must be, in addition to the restriction mentioned above, a non-ambiguous means of distinguishing between the members of the list and any following forms.

Two prefixes are allowed on either basic or compound forms. An Optional prefix specifies that the given form need not appear. That is, if the corresponding argument does not match the form, then that form may be skipped, and processing continued by matching the same argument with the next form in the list. It must be possible to distingusih between the optional form and any following (optional) forms. A Repeat prefix specifies that the form given may appear any number of times and that the body of the procedure is to be executed once for each time the form appears, and after all variable assignment for each match have taken place. There may be no nested Repeat specifications.

One final mechanism is provided to allow a list or string to be built from several arguments. For contiguous arguments, the syntax is:

```
Form {List|String} <identifier>
```

which forms a list or string out of all arguments up to but not including the first which matches the next specified form (or the end of the argument list). This next form, which acts as a delimited for the list or string, must be a keyword. Once the string or list has been built, it is assigned to the <identifier> interpreted as a <reference>. The value may also be built from the occurences of certain noncontiguous arguments. The syntax for this variant is:

```
Form {List|String} <identifier>
     From <parm spec>
```

Any explicit assignment to the <identifier> within the <parm spec> will instead add a new element to the list or string. That is, an assignement of the form:

```
Let <identifier> = <expression>
```

will become for the list form of a <construction>

```
Let <identifier> = <identifier> ! <expression>
```

(For strings, each new element is converted to a string and concatenated to the end of the existing string along with one intervening blank.)

Two basic forms are distinguishable if they are both keyword specifications or if one is a keyword and the other is a type or reference specification. Two groups are distinguishable if their first forms are distinguishable.

Semantic Meaning

There are two means available to provide semantic information. First, each compound or basic form may be immediately followed by a <success> specification, an <executable unit>, to be executed if the form is present. For example:

```
Select
     Keyword -working_dir Or -wd
          Let Dir = WorkingDir
     Pathname Dir
End
```

Second, each compound or basic form may be followed by a <default> specification, an <executable unit>, to be executed if the form was allowed to not appear (i.e. optional or appearing in a Select or Multiple form) and did not indeed appear. For example:

```
Optional Select
     Keyword -brief Or -bf
          Let BriefSw = True
     Keyword -long Or -lg
          Let BriefSw = False
End
Default Let BriefSw = False
```

Error Processing

If the arguments supplied to a command do not correspond to what is required for the command, an error message will be generated automatically. The message is selected from the following:

1. Too many arguments. After processing the last expected argument, there exist as as yet unscanned ones.

2. Bad syntax in command. A required Keyword is missing. (A check is made to see if the next form is what it should (could) be.)

3. Expected argument missing. A required argument, as in a type or reference form, is not present. (A check, as above, is made.)

4. Expected argument group missing. An entire group or construction is missing. The pharse "argument group" is replaced by a <title string> if one is specified for the group.

5. Argument is not convertible to <type>: <arg>. A required argument is not of the designated type. (A check is perform to determine that an argument is present in that location, and not just missing.)

6. Extraneous argument present. An extra argument is present, that is, the next argument is what the curent one should be.

7. Invalid keyword; <keyword> expected. Issued when an invalid keyword appears in the place of a required or optional keyword.

8. Invalid syntax in argument group. A required group or optional group whose first members have been matched contains unmatchable forms. The pharse "argument group" may again be replaced by the <title string> for the group.

9. Invalid option. There are argument(s) present that do not match any form in a Select or Multiple specification.

10. Invalid syntax in command. Issued when all else has failed.

11. Invalid syntax in command; arguments <arg1>...<argn> not recognized. Issued in the above case, but when later forms can be matched.

The user can specify the action to be taken if one of the errors, 2, 3, 4, or 5, occur by using an <error> statement. For basic forms, the statement supplied is executed if the form was required to appear but did not. Note that an <error> neec not be applied to forms within a Select or Multiple form, and also that <default> and <error> statements are mutually exclusive.

Order of Processing

A <parameter block> is interpreted by performing the following operations in the order indicated:

1. The arguments passed to the <procedure> or <on unit> are scanned and matched according to the rules given for each form. If an error is detected, then the <error> action or default error action is taken as applicable, and processing aborted.

2. All implied assignments (as for the type and value form) are performed in unspecified order.

3. The <default> actions for optional forms that did not appear are executed in unspecified order.

Upon completion, the rest of the body of the <block> is executed. (That is, step 6 of block activation is performed.) If there are additional groups of arguments to be processed for a repeat form, steps 2 and 3 are repeated for only those forms that appeared in the repeated <parm spec>s. The body of the <block> is also rexecuted.

BUILTIN FUNCTIONS, IDENTIFIERS, AND PSUEDO-VARIABLES

A number of computational and special purpose functions are provided by the language. These functions may be invoked by name in the same manner that a user defined procedure would be, provided that the user has not defined one with the same name, and if Builtin is specified as part of the current environment.

The description of these functions will include their name, parameters, and the type of their result. Most of the functions require that their parameters be of a specific type. If an argument is not of the correct type, it will be converted or promoted as appropriate. The type expected for a parameter will be designated by the letter denoting the parameter:

a	arithmetic (real or integer)
b	branch
d	date
i	iocb
l	list
p	pathname
r	reference
s	string
t	logical
v	value (anything)
x	address

The result is indicated by "->" (which may be read as evaluates to) followed by a type letter. Psuedo-variables are indicated by "<->" instead of "->"; in all cases, the type of the assigned value is the same as the result.

Arithmetic Builtins

These perform the same function as their counterparts in PL/I. The operands must be (convertible to) arithmetic values. The result is either an integer or real value depending on the precision needed to express the result. The functions provided are:

1. Mod a1 a2 -> a

2. Min a1 ... an -> a

3. Max a1 ... a2 -> a

4. Ceil a1 -> a

5. Floor a1 -> a

6. Abs a1 -> a

Type Conversion Builtins

These may be used as both functions and psuedo-variables. When used as a function with one argument, they convert the value of the one argument to the type implied. When used as a function of no arguments, they return a value of the specified type for use in type comparisions. When used as psuedo-variables, they take the value assigned to them, convert it, and assign it to their one argument. There is one such builtin for each data type. Conversions are performed in the manner described in he section on conversions.

String Builtins

The first group of string builtins perform the same function as their PL/I counterparts. Provided are:

1. Index s1 s2 -> a

2. Substr s1 a1 [a2] <-> s

3. Reverse s1 -> s

4. Verify s1 s2 -> a

5. Search s1 s2 -> a

6. Length s1 -> a

The second group of string builtin perform certain special functions. Specifically:

1. Suffix r1 s1 <-> s

   Appends a suffix given by s1 to (String r1) if the suffix is not already present. For example:

(Suffix "x" ".pl1") -> "x.pl1"
(Suffix "x.pl1" ".pl1") -> "x.pl1"

This may also be used as a psuedo-variable to assign a string, guarenteed to contain the specified suffix, to r1:

Let Suffix [A] ".pl1" = "x"

sets A to the value "x.pl1". This is particularly useful for Pathname parameter specifications.

2. Strip r1 s1 <-> s

This removes a suffix given by s1 from (String r1) if the suffix already appears. For example:

(Strip "x.pl1" ".pl1") -> "x"
(Strip "x" ".pl1") -> "x"

This may also be used as a psuedo-variable to assign a string, guarenteed not to contain a suffix, to r1:

Let Strip [A] ".pl1" = "x.pl1"

sets A to "x".

3. Format s1 v1 ... vn -> s

This returns a string which is the result of editing the values of r1 through rn under control of the ioa_ style format s1. For example, if A = 1.23, then:

(Format "A = ^d" A) -> "A = 1.23"

List Builtins

The list builtins may be divided into two groups. The first are functions which perform the same sort of operations on list elements as the string builtin functions do for characterers.

1. Index l1 l2 -> a

Example: (Index {{1 2 4 5}} {{2 4}}) -> 2

2. Reverse l1 -> l

Example: (Reverse {{1 2 3}}) -> {3 2 1}

3. Verify l1 l2 -> a

Example: (Verify {{"5" "4" "6"}} {{5 7}}) -> 2

4. Search l1 l2 -> a

   Example: (Search {{yes no}} yes) -> 1

5. Length l1 -> a

   Example: (Length {{1 True a B}}) -> 4

The second group performs certain special functions:

1. Expand l1 -> l

   This performs "iteration" processing on its argument. The result is a list formed by concatenating together corresponding elements of the first level sublists and scalars. All first level sublist must be of the same length. For example:

   ```
   (Expand {{1 {2 3} 4 {5 6}}}) ->
                              {{1 2 4 5} {1 3 4 6}}
   ```

2. Eval s1 -> l

   This returns a list which is the result of tokenizing and evaluating the contained expressions:

   ```
   (Eval "a (2 + 2) -c") -> {"a" 4 -c}
   ```

Input/Output Builtins

1. Line [i1] <-> s

   This may be used as function or psuedo-variable to read or write one line to the I/O switch specified by i1 (defaulting to user_input or user_output). Examples:

   ```
   (Line) -> one line of input including <NL>

   Let Line error_output = "help"
   ```

   writes "help" || <NL> on the switch error_output.

2. Input [i1] -> s, Output [i1] <- s

   These perform the same function as Line except that they read (write) one token or expression from (to) the designated switch.

3. Query s1, Response s1 -> s

These ask the question given by s1 and return the string containing the answer given by the user. Query restricts the answer to being either yes or no.

4. UserInput, UserOutput, ErrorOutput -> i

These represent the IOCB's of the corresponding I/O switches.

Argument Builtins

1. Argument -> l

This identifier is bound to the argument to the argument list with which a procedure, function, or psuedo-variable was invoked. It may be used as a list value would.

2. Narguments -> a

This identifier is bound to the value of (Length Argument).

3. Target -> l

This returns a logical value indicating whether or not a psuedo-variable was invoked as the target of an assignement. If invoked from other than a psuedo-variable, it returns the value Undefined.

4. AssignedValue -> v

This identifier is bound to the value that was assigned to a psuedo-variable. It is an error to reference this function in a context where Target yields a value other than True.

Miscellaneous Builtins

1. True, False -> t

These are bound to the logical values true and false.

2. Null -> x

This is bound to the null address value.

3. Undefined

This returns an undefined value -- that is, the value of the object to which a newly defined identifier is bound.

4. Converts r1 r2 -> t

This returns a logical value indicating whether or not r1 can be converted to the type of r2. Example:

(Converts 2 Address) -> False

Segment_Name_Builtins

1. Directory p1 -> p

This returns the pathname designating the parent of the entry designated by p1. The parent of the root is itself.

2. EntryName p1 -> s

This returns the character string giving the entry portion of the pathname, p1. The entry name of the root is "".

3. HomeDir <-> p

This psuedo-variable represents the pathname of the user's home directory -- or default working directory.

4. WorkingDir <-> p

This represents the pathname giving the user's current working directory.

5. Unique -> s

This returns a character string containing a unique character string.

6. Segments [p1], Directories [p1], Links [p1], MSFs [p1], Files [p1] -> l

These return a list of branch values identifying directory entries whose names match the (star laden) pathname given by p1 and of the approriate type. (Files include segments and multisegment files.) The default value for p1 is WorkingDir || ">**".

7. Match s1 s2, MatchPath s1 p1 -> t

These return a logical value indicating whether or not the string s2 (EntryName p1) matches a given starname, s1. For example:

(Match *.*.archive c.archive) -> False
(MatchPath *.*.archive <tools.s.archive) -> True

8. Equal s1 s2 -> s, EqualPath p1 s2 -> p

These implement the equal convention. The first parameter represents the source string; the second, the equal pattern with which to edit the first. The result of Equal is the string giving the edited name. EcualPath uses the entry portion of the pathname, but the result has the directory portion restored. For example:

(Equal prog.s.archive =.archive) -> "prog.archive"
(EqualPath >udd>p>pers>x.pl1 a.=) ->
">udd>p>pers>a.pl1"

Branch Builtins

These functions return information about the attributes of a specified directory entry. For those attributes for which it is sensible for the user to alter the values, they may also be used as psuedo-variables.

1. Author b1 -> s

2. BCAuthor b1 -> s

3. Dtm b1, Dtu b1, Dtd b1, Dtem b1 -> d

   These return the date/time modified, date/time used, date/time dumped, and date/time entry modified respectively.

4. Type b1 -> s

   This returns either "segment", "directory", "link", or "multisegment file".

5. CurrentLength b1 -> a

6. RecordsUsed b1 -> a

7. LinkTarget b1 -> p

   This returns the pathname that a link points to.

8. NullLink b1 -> t

   This returns a logical value indicating whether a link points to an existing branch.

9. BitCount b1 <-> a

10. CopySwitch b1 <-> t

11. SafetySwitch b1 <-> t

12. Quota b1 <-> a

13. MaxLength b1 <-> a

14. ACL b1, IACL b1 <-> l

    Where the list takes on the form:

    { {<mode> <aclname>}... }

15. Names b1 <-> l

    where l is a list of string values giving the names on the entry.

16. RingBrackets b1 <-> l

    where l is a list of length three (for segments) or length two (for directories) containing integer values giving the ring brackets on the entry.

If the entry specified by b1, does not have an attribute of the particular type specified, then the program is in error.

EXAMPLES

The following is an example of a very simple command. It performs the function of the current add_name command.

```
Procedure add_name
     Entry an

     Temp Code

     Parameter BranchName, NewName
          Pathname Branchname
          Repeat String NewName
     End

     hcs_$chname_file (Directory BranchName)
                    (EntryName BranchName) "" NewName [Code]
     If Code ^= 0 Then Do
          com_err_ Code "add_name" NewName
          Return
     End
```

```
    End
```

The following subroutine is intended to invoked the first thing in a process. That is, it is equivalent to a current start_up.ec.

```
Procedure StartUp

     Global LastLogon

     Parameter Interactive, NewProc
          Optional Keyword -login
               Let NewProc = False
               Default Let NewProc = True
          Optional Keyword -absentee
               Let Interactive = False
               Default Let Interactive = True
     End

     If Interactive Then Do
          accept_messages -print -brief
          mail -brief
          If ^ NewProc Then Do
               memo -brief
               Do I From Segments >doc>info>*.info
                    If LastLogon < Dtm I
                         Then Line = Entry I || "modified."
               End
          End
          LastLogon = Clock
     End

End
```

It peforms more or less standard functions on login and new_procs. In addition, it examines all info segments to see if any have been modified since the last login.

The following two subroutines show the use of the condition mechanism in ECL with the command query mechanism. First, is an example of how the built in function query could be coded.

```
Function Query Question

     Temp Answer

     Signal command_query_ (String Question) True [Answer]
     Return

End
```

Second, is an example of how to code the answer command.

```
Procedure answer

     Parameter BriefSw, Answer, Command
          Value Answer
          Optional Keyword -bf Or -brief
               Let BriefSw = True
               Default Let BriefSw = False
          Form List Command
     End

     On command_query_ Question YesNoSw [Ans]
          If ^ BriefSw
               Then Line = Question || " " || Ans
          If YesNoSw
               Then Do While Verify Ans {yes no} > 0
                    Output = "Please answer ..."
                    Ans = Input
               End
     End

     (Command 1) (Command 2 (Length Command - 1))

End
```

The following subroutine performs essentially the same function as the current Multics command processor.

```
Procedure command_processor_ CommandLine

     Environment External
     Temp CommandLine
     Builtin Expand, Length, Eval

     Do Command From (Expand (Eval CommandLine))
          (Command 1) (Command 2 (Length Command - 1))
     End

End
```

The command line to be interpreted is passed as an argument to the procedure. The function, Eval, is invoked to parse the line into basic expression and evaluate them (by reference). Iteration processing is then performed by Expand. Each resulting command is then invoked, one at a time, with the Call statement embedded in a list iteration loop. The environment statement insures that only external commands (not builtins or procedures defined in a calling block) will be found by the Call. Note that specifying just "Environment" would, in a like manner, restrict the user to calling procedures defined in the same subroutine. In this way, a restricted subsystem or language interpreter could be constructed.

The following is an example of the use of block structure to control the environment. It is the command walk_subtree written in a manner to exploit a local copy of the variable working directory.

```
Procedure walk_subtree
     Entry ws

     Parameter Dir, Command, Brief
          Select
               Keyword -wd Or -working_dir
                    Let Dir = WorkingDir
               Pathname Dir
          End
          Optional Keyword -brief Or -bf
               Let Brief = True
               Default Let Brief = False
          Form List Command
     End

     Walk Dir
     Return


     Procedure Walk Wdir
          Local WorkingDir = Pathanme Wdir

          If ^ Brief Then Line = WorkingDir
          (Command 1) (Command 2 (Length (Command - 1))

          Do I From Directories
               Walk I
          End

     End

End
```

When the command gets executed within the the internal procedure, walk, the local variable WorkingDir has been set. If the command procedure called makes use of the variable explicitly (and found through the Caller, not Builtin, rule) then the correct things will happen. Correct functioning is also dependent on the builtin Directories and the conversion process also being aware of the new copy of the variable. This approach is particularly desirable as the change is local; with the environment for command level set to exclude calling blocks, a new command level created as the result of a quit signal is isolated from the changing state of the current working directory which may be ongoing in the previous level.

The following two examples show alternate ways in which a "default" value can be obtained for an argument. In the first case, prompting is used to acquire the missing value.

```
Procedure pl1

     Parameter SourceFile, Map, ...
        Optional Pathname (Suffix SourceFile ".pl1")
            Default Do
                Line ErrorOutput = "Enter source file -"
                Pathname (Suffix SourceFile ".pl1") = Input
            End
        Optional Multiple
            Keyword -map
                .
                .
        End
    End

    ...

End
```

In the second example, a global default value is used. This default value maintains the idea of a current file name which may be used in or set by any command that uses it.

```
Procedure pl1
    Global CurrentFile

    Parameter SourceFile, Map, ...
        Optional Pathname (Suffix CurrentFile ".pl1")
            Default Let Line = "Assuming " || CurrentFile
                                          /* CurrentFile is global */
        Optional Multiple
            Keyword -map
                .
                .
        End
    End

    ...

End
```