1. PURPOSE AND FUNCTION

The primary purpose of the dictionary relational expression language, dREL, is to enable relationships between data items in a dictionary to be specified, simply and succinctly, as a symbolic methods script written in dREL. The facility to derive data values from other items provides a powerful approach for precisely defining data and mitigates against the need to archive derivable tertiary data, and much of the secondary data - as these can now be calculated from the primary data present in data files.

The definition example in Fig 1 shows how dREL methods are used. This definition contains the attribute _method.expression which specifies, in dREL, the crystallographic unit cell volume as a function of the cell lengths and angles.

save_cell.volume
   _definition.id            '_cell.volume'
   _description.text
   ;
   Volume of the crystal unit cell.
   ;
   _name.category_id         cell
   _name.object_id            volume
   _type.container            Single
   _type.contents             Real
   _type.purpose              Measurement
   _enumeration.range         0.0:
   _units.code                angstroms_cubed
   _method.expression
   ;
   With v as _cell_vector
   _cell.volume = v.a * ( v.b ^ v.c )
   ;

Figure 1: Definition of the crystal cell volume.

The evaluation process works as follows, assuming that a data file is being read with a search utility that uses associated domain dictionaries for validation and checking support. If the item _cell.volume is requested but its value is not present in the file, the utility automatically transfers the script from _method.expression to a dREL handler. This parses the script, identifies the length and angle items needed to evaluate the cell volume, requests these values from the data file, and calculates the volume. The evaluation process assumes that any data item referenced not in the data file will itself be derived from a methods expression. The dREL parser will recursively derive data values as needed, until either the required items are found or calculated, or the relationship pathways are exhausted. The calculated cell volume is passed back to the utility, which responds identically to the request as if the value had been present in the data file.

This example shows that methods expressions in the dictionaries provide a clarity and precision not achievable in the past. The use of methods, with the coalescence of dictionaries, will promote an exploitation of data well beyond that achievable in the past. For example it would mean that only primitive data need be archived in data files, and the related data can be derived when needed using algorithms contained in the dictionary. This would reduce the amount of data that needs to
be exchanged and archived. Some derived quantities (e.g. atomic coordinates), may continue to be archived, but, having the methods definitions in associated dictionaries, specifying precisely how they were derived, will enable new derivations to be evaluated as better approaches are developed.

2. PRIMITIVE DATA TYPES

dREL supports the following primitive data types of the values for variables appearing in methods expressions. Local variable names (as opposed to global data tags) are restricted to alphanumeric characters only.

- Character strings
- Integer numbers
- Real numbers
- Complex numbers
- Measured numbers

Data typing may be achieved by explicitly within the dictionary definitions of the object, or implicitly from usage in an expression, or explicitly using a function. DDLm dictionary definitions specify data types using the TYPE attributes (see _type.contents, _type.container, _type.purpose, _type.dimension).

2.1 CHARACTER STRINGS

2.1.1 Dictionary definition

The data dictionary specifies the type of a data tag using the TYPE attribute _type.contents.

2.1.2 Inline definition

Character strings are created by enclosing a string in quoting literals. Matching single and double quote characters at the extremities of a single line string implicitly identify a literal object as TYPE CHARACTER. Matching triple quote characters at the extremities of a multi-line string implicitly identify a literal object as TYPE CHARACTER.

2.1.2.1 Single quotes

Matching single quote characters at the extremities of a single line string implicitly identify a literal object as TYPE character. The following is simple character string.

'single quotes make it easy to embed a "double quote"

2.1.2.2 Double quotes

Matching double quote characters at the extremities of a single line string implicitly identify a literal object as TYPE character. The following is simple character string.

"double quotes make it easy to embed a 'single quote"

It is also possible to use C-style elides to achieve this effect.

"double quotes don’t prevent the use of a \"double quote\"

2.1.2.3 Triple quotes

Matching triple quote characters at the extremities of a multi-line string implicitly identify a literal object as TYPE character. The following is simple character string.

"" triple quotes
are
multi-line""
This is equivalent to
"triple quotes
are
multi line"

Triple quotes comprised of the single quote literal are also supported.
''single or double quotes are can be
used to define the triple quote sequence.''

2.1.2.4 Special explicit strings

$dREL$ provides for two special string literal definitions; $raw$ and $Unicode$ strings.

A raw string is delimited by $r"..."$. Characters in a raw string are interpreted literally and
regular expressions or sequences of characters are protected from parser interpretation. Here is
an example.
r"raw quotes don't interpret escapes viz:\n<< not a newline!"

This is equivalent to the following string:
"raw quotes don't interpret escapes viz:\n<< not a newline!"

2.2 INTEGER NUMBERS

$dREL$ supports decimal, binary, octal and hexadecimal Integer numbers. These are identified in
three ways; explicitly from dictionary definitions of the object, implicitly from usage in the
expression language, or explicitly using a function.

2.2.1 Dictionary definition

The data dictionary specifies the type of a data tag using the TYPE attribute _type.contents.

2.2.2 Inline definition

2.2.2.1 Decimal integers

The syntax of a decimal integer is: $[+-]?[0-9]+$
An example decimal integer is: -23

2.2.2.2 Binary integers

The syntax of a binary integer is: $[0][bB][0-1]+$
An example binary integer is: 0b1101110010111000

2.2.2.3 Octal integers

The syntax of a octal integer is: $[0][oO][0-7]+$
An example octal integer is: 0o63103

2.2.2.4 Hexadecimal integers

The syntax of a hexadecimal integer is: $[0][xX][0-9a-fA-F]+$
An example hexadecimal integer is: 0x6672af

2.3 REAL NUMBERS

$dREL$ supports decimal and scientific Real (or floating-point) objects. Real numbers are identified in
three ways; explicitly from dictionary definitions of the object, implicitly from usage in the
expression language, or explicitly using a function.

2.3.1 Dictionary definition

The data dictionary specifies the type of a data tag using the TYPE attribute _type.contents.
2.3.2 Inline definition

3.3.2.1 Decimal real numbers

The syntax of a decimal real number is: \( [+\text{?} ([0-9]+.[0-9]* | .[0-9]+) [[Ee][+\text{?}][0-9]+]? \)

An example decimal real number is: -7893.8221 or -7.89382e+3

2.3.3 Explicit definition

Conversion to real number is achieved with the function:

- \( \text{Float()} \)

2.4 \textit{COMPLEX NUMBERS}

dREL supports Complex number objects. Complex numbers are identified in three ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

2.4.1 Dictionary definition

The data dictionary specifies the type of a \textit{data tag} using the TYPE attribute \_type.contents.

2.4.2 Inline definition

2.4.2.1 Complex numbers

The syntax of a complex number is: \( ((\text{Real} | \text{DecimalInteger}) [+\text{?}]) (\text{Real} | \text{DecimalInteger}) [j] \)

An example complex number is: -7893.8221+54.924j

2.4.3 Explicit definition

Conversion to a complex number is achieved with the function:

- \( \text{Complex (Nreal, Nimag)} \)

2.5 \textit{MEASURED NUMBERS}

A \textit{Measured value} consists of a number and its standard uncertainty appended in parentheses. The uncertainty value is an integer scaled to the precision of the last digits of the measurement value. \textit{Measurement} numbers are identified in three ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

2.5.1 Dictionary definition

The dictionary definitions declare the TYPE of a data tag with the following set of attribute declarations:

- \_type.contents
- \_type.purpose

The value of the attribute \_type.contents can also be \textit{Integer} or \textit{Complex}.

2.5.2 Inline definition

3.5.2.1 Measured numbers

The syntax of a measurement number is: \( ([\text{Real} \backslash \text{DecimalInteger}] \backslash ([1-9][0-9])* \)

An example measurement number is: -783.2(12) = -783.2±1.2

Other examples are x.xxE-yy.zz or x.xx(zz)E-yy or x.xxE-yy(z.zzE+ww) where a ‘.’ in the standard uncertainty value indicates an explicit value.
2.5.3 Explicit definition
Conversion to a measurement number is achieved with the function:

- \textit{Measure}(val, su)

3. CONTAINER TYPES FOR \textit{dREL}

\textit{dREL} supports the container types

- List
- Tuple
- Table
- Array
- Single

\textit{dREL} also supports the nesting of container types i.e. the definition

\begin{verbatim}
  _item.container         Tuple[Tuple]
  _item_dimension        [5[3]]
\end{verbatim}

refers to a list of five tuples each containing three elements.

3.1 \textit{LIST} CONTAINERS

\textit{List} containers are mutable objects with the following properties.

- \textbf{Type}: contained items may be of any, but the same, \textit{TYPE}.
- \textbf{Dimension}: Lists are single dimensioned.
- \textbf{Size}: the length of a table is not pre-defined.
- \textbf{Access}: indexed by integers (implied starting index is 0).
- \textbf{Shape}: may be nested.

Lists are created in three ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

3.1.1 Dictionary definition
The dictionary definitions declare the nature of a \textit{List} container with attribute declarations. Here are such declarations for a list of real numbers of nine elements.

\begin{verbatim}
  _type.container              List
  _type.contents               Real
  _type.dimension              [9]
\end{verbatim}

3.1.2 Inline definition
Lists may be defined inline using the \textit{List(...)} function. E.g.

\begin{verbatim}
  List ([1, 7, 3, 10])
\end{verbatim}

3.2 \textit{TUPLE} CONTAINERS

\textit{Tuple} containers are immutable objects with the following properties.

- \textbf{Type}: items may be of any \textit{TYPE}.
- \textbf{Dimension}: are single dimensioned.
- \textbf{Size}: needs to be defined.
- \textbf{Access}: indexed by integers (implied starting index is 0).
- \textbf{Shape}: may be nested.
Tuples are created in three ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

### 3.2.1 Dictionary definition

The dictionary definitions declare the nature of a Tuple container with attribute declarations. Here are such declarations for a tuple of three values.

```plaintext
_type.container               Tuple
_type.dimension              [3]
```

### 3.2.2 Inline definition

Tuples may be defined inline using the `Tuple(...)` function. E.g.

- `Tuple (10.2, 12.3, 7.4)`
- `Tuple ('a', 'b', 'static')`

### 3.3 Table Containers

Table containers are similar to Lists except that each value in the table may have an associated key. A table has the following properties.

- **Type:** can contain values of any, but the same, TYPE.
- **Dimension:** single dimensioned list; each "key":val is considered as one element.
- **Size:** the length of a table is _not_ pre-determined.
- **Access:** by key; the default keys are sequential integers starting at 0.

Tables are created in two ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

#### 3.3.1 Dictionary definition

The dictionary definitions identify a Table object with the following attribute declarations.

```plaintext
_type.container               Table
_type.contents                Real
```

A Table differs from a List (see §3.1) in several important ways. A List object contains a specified number of values that are identified explicitly by sequence. A Table contains a sequence of character or number values which identified by a key.

#### 3.3.2 Explicit definition

Conversion of a sequence of objects to a new list is achieved with the function:

- `Table (‘key’:val,..)`
- `Table (“left”:“links”, “right”:“recht”)`

### 3.4 Array Containers

Array containers are immutable objects with the following properties.

- **Type:** only contain items of number TYPE.
- **Dimension:** are single/multi- dimensioned.
- **Size:** pre-defined upper and lower extents.
- **Access:** indexed by integers.

Arrays are created in two ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

#### 3.4.1 Dictionary definition
The dictionary definitions declare the nature of an array with attribute declarations. Here are the attributes for defining a three element integer vector, indexed from 0 to 2.

<table>
<thead>
<tr>
<th>_type.container</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>_type.contents</td>
<td>Integer</td>
</tr>
<tr>
<td>_type.dimension</td>
<td>[3]</td>
</tr>
</tbody>
</table>

### 3.4.2 Inline definition

Vectors may be defined inline using the `Array(....)` function. E.g.

```
Array ([10.2, 12.3, 7.4])
```

### 3.5 SINGLE CONTAINERS

Single containers are a single value with the following properties.

- Type: may be of any TYPE.
- Dimension: a single value.
- Size: 1.

Single values are created in three ways; explicitly from dictionary definitions of the object, implicitly from usage in the expression language, or explicitly using a function.

#### 3.5.1 Dictionary definition

The dictionary definitions declare the nature of a Single container with attribute declarations. Here is a declaration for a real number.

<table>
<thead>
<tr>
<th>_type.container</th>
<th>Single</th>
</tr>
</thead>
<tbody>
<tr>
<td>_type.contents</td>
<td>Real</td>
</tr>
</tbody>
</table>

#### 3.5.2 Inline definition

Single values may be specified inline by equating it to another single value. E.g.

```
a = 5.
Z = a
```

## 4. LANGUAGE BASICS

In this section the basic syntax of \textit{dREL}, and the language elements that lead up to controlling the execution flow, are introduced. It is important to appreciate that \textit{dREL} does not support, or require, data declarations other than those already discussed in §3. Nor does it provide, in this version at least, input/output control statements.

### 4.1 ASSIGNMENT EXPRESSIONS

#### 4.1.1 Named objects

A NAMED object or “variable” in \textit{dREL} may only be created on assignment (see §4.1.2). The typing of a variable is by coercion (see §4.1.3 and §4.2). The scope of a variable is local.

#### 4.1.2 Assignment statement

The process of object transfer is initiated with the “=” character which transfers the value of the right-hand expression of objects \textit{Robjects} to the left-hand objects \textit{Lobjects}. The general form of the object transfer is:

```
Lobjects = Robjects
```
In the example below the value of the literal Integer object, “5”, is assigned to a mutable NAME object, the variable string “x”.

```
x = 5
```

Objects may also be an expression of objects.

```
x = y * z
y = Sin (a) + Cos (a)
```

Multiple transfers are also allowed.

```
a, b, c = 3.628, -7.67, 5.329
```

### 4.1.3 Assignment TYPE

In dREL, object types are not declared. We have already seen in §3, the typing of Objects items may be determined from dictionary definitions, inline typing constructions or simply inferred by association with objects of known type. The TYPE of LObjects may be set by the same mechanisms, or result directly from the inferred type of the Objects value.

It follows that the statement

```
x = 5
```

sets the TYPE of “x” as Integer. A new assignment of “x” in the next statement

```
x = 10
```

is permitted because it has a consistent TYPE. However, the assignment

```
x = "Hello World"
```

is illegal but will not cause an error message to be raised.

This is contrary to the practice of some scripting languages, but it avoids the faulty and misleading construction of expressions.

### 4.2 TYPE COERCION RULES

Type coercion rules are needed when Objects expressions contain objects of mixed type. dREL uses the following coercion rule, in order of increasing priority.

```
Integer → Real → Complex
```

In the next statement, LObjects is of type Real, provided this is the first assignment to "x".

```
x = 5 + 7/2
```

### 4.3 COMMENTS

Comments are non-executable strings. In dREL a sequence of characters following an unquoted sharp or hash symbol # is interpreted as a comment, up to the end-of-line character. Here are typical examples.

```
x = 5     # a comment follows an in-line hash
```

The following statement does not contain a comment because the hash symbol is contained within a quoted string.

```
s = "# this is *not* a comment"
```

### 4.4 EXPRESSION OPERATORS AND TERMINATORS

dREL supports the following arithmetic expression operators
addition
* product (dot product when applied to vectors)
^ cross product of vectors
** power of
- subtraction
/ division

The operands apply to Integer, Real and Complex number objects. They are also applicable to the containers List, Tuple, Table, and Array provided the elements of these are of TYPE number. The expression operators + and * have meaning for manipulating character strings.

dREL supports the following logical expression operators
== equals
!= not equals
> greater than
< less than
>= greater than or equals
<= less than or equals
and and
or or
not not
in matches element of the list
not in does not matches element of the list

dREL supports the following expression terminators
; semicolon separates multiple expressions in a line
\n newline closes a line unless a balancing ‘)’, ‘}’ or ‘]’ is missing

Example statements using these terminators follow.

\n  a = 234 ;  y = 45 ;  z = -2
\n  b = (y + z)/2.0
\n  c = (45 + 72 * (93 + 4) + z)

4.5 Supported Escape Sequences

The following special escape character sequences are supported in dREL expressions. Note that the same diagraphs may be used for other purposes in data values, but within the literal dREL scripts the following meanings will be assumed.

\n newline
\r carriage return
\f formfeed
\t horizontal tab
\b binary bit pattern # implements backspace!
\o octal bit pattern # try \0xxx instead
\x hexadecimal bit pattern
\0 null character
\ \ backslash (\)
\u Unicode character in hexadecimal E.g. \u0022 == "

Note that a Unicode character in a string makes the entire string of TYPE Unicode.
5. FLOW CONTROL

*dREL* supports a range of standard and specialised flow control statements and terminators for controlling the repeated execution of object expressions. These are as follows:

- Indexed Do
- List Repeat
- List For
- List Loop
- List With
- List Where
- List Break
- List Next
- If/ElseIf/Else
- Switch/Case/Default

The essential constituents of a repetitive execution sequence, is as follows.

```
repeat-statement

  *expression block*

repeat-terminator (optional)

```

If more than one expression exists within the expression block, it MUST be enclosed within a set of braces "{" and "}". If only one expression is repeated, its association with the *repeat_statement* is implied and the braces are optional. In general, it is good and safe programming practice to always use braces to bound the repeated expression block.

### 5.1 DO STATEMENT

Indexed repetition of expressions is supplied with a *Do* statement.

```
Do  index = first, last, incr  
    *expression block*  
    repeat-terminator (optional)  
```

The *index* variable is initialised with the *first* index value (or variable) and executes the expression block provided index is less than or equal to the *last* index value (or variable). The *index* is incremented by the *incr* value AFTER each execution of the expression block. The *incr* value is option and has a default value of 1.

A typical application of the Do operator follows.

```
Do i = 0, 20, 2  
    total = total + subtotal[i];  
```

### 5.2 REPEAT STATEMENT

Unindexed repetition of expressions is supplied with a *Repeat* statement.

```
Repeat  
    *expression block*  
```

The expression block MUST contain one or more invocations of the Break statement in order to exit the repeat loop. Repeat loops may be nested. A typical application of the Repeat operator follows.

```
Repeat  
    {  
        i=i+1;  
        if(i>100) Break;....  
    }  
```

### 5.3 FOR STATEMENT
Manipulation of List items is provided with a *For statement.*

```
For a in list : n op m  { * expression block * }
```

where `a` is the current element of the entire `list`. An optional expression “:n op m” is available to control the accessing of list packets, where `n` is the index (starting at 0) for each packet; `op` is the test operator (`<> <= >=` allowed) and `m` is the test integer operand. The `op` and `m` entries are optional. The index `n` is a local variable and may be tested elsewhere in the script.

An example where `list` is a literal object follows.

```c
i = 0
For a in ["Mon","Tues","Wednes","Thurs","Fri"] {
    Day[i] = a + "day"; i += 1;
}
```

### 5.4 LOOP STATEMENT

A fundamental function of *dREL* is to apply and derive data in a *data file* using definitions in a dictionary. Much of this data is in looped lists, and, consequently, there needs to be a simple and transparent way to identify and apply repetitive data items. Data items in the same list are, according to the dictionary language *DDLm*, classified as belonging to the same generic category group. The id code of a category is therefore a convenient tag to identify groups of items, and to access “packets” (i.e. sub-lists) of data items in lists. The Loop repetition operator is provided primarily for this purpose.

```
Loop local as list : n op m  { * expression block * }
```

The string `local` is an object variable, local only to the specific methods script in which it is invoked, which assumes the successive values of `list` during the repeated execution of an expression block. If `list` is a category id code, then the `local` object contains successive sub-list of tagged values (i.e. an implicit Table) and individual data items may be accessed as *object attributes* of `local`. An optional expression “:n op m” is available to control the looping of list packets, where `n` is the loop index (starting at 0) for each packet; `op` is the test operator (`<> <= >=` allowed) and `m` is the test integer operand. The `op` and `m` entries are optional. The index `n` is a local variable and may be tested elsewhere in the script.

#### 5.4.1 Data Loop Example 1

A simple invocation of *Loop* will now be considered for data. This example will access two data items in the category *POSITION*, known by their data names as *_position.vector_xyz* and *_position.object_id*. An abbreviated definition of the category and these items follow. Note that *_position.object_id* is specified as the category key to each packet of these items.

```
_category.id                 position
_category_key.index_id        '_position.object_id'

_definition.id               '_position.number'
[name.category_id            position
[name.object_id               number
_type.container              Single
_type.contents               Integer
_type.purpose                Index

_definition.id               '_position.object_id'
[name.category_id            position
[name.object_id               object_id
_type.container              Single
_type.contents               Uchar
```
In a data file these items might appear in a looped list (abbreviated) as follows.

```
loop_
  _position.object_id
  _position.number
  _position.vector_xyz
  1   origin   [0.0, 0.0, 0.0]
  2      body-diagonal   [5.0, 5.0, 5.0]
  32  diagonal-terminal   [10.0, 10.0, 10.0]
```

In a dREL script the Loop construct allows individual items in a packet (in this instance the packet contains three values) to be addressed by the extension name defined in the dictionary with the attribute _item.extension (i.e. number, object_id and vector_xyz).

```
Loop a as position {
  If (a.object_id == "origin") {
    CoordOrigin = a.vector_xyz
  }
  Else
    LocalPosn[a.number] = a.vector_xyz
}
```

### 5.4.2 Data Loop Example 2

Another example is needed to illustrate the functionality of the Loop operator when handling lists of data from non-hierarchically-related but derived categories. The prototype dictionary language allows hierarchical relationships between data items to be defined, via category definitions, and these provide access "pathways" which are independent of how these related data are stored in the data file. For instance, items in the same category, or in hierarchically-related categories, may be accessed as an attribute extension of either the name of the “parent” category (i.e. the highest category in the family hierarchy) or the name of the hierarchically-related category.

All data in a looped list be of the same category family. Items from hierarchically-related categories may be in more than one looped list but for the purposes of access, the dREL parser subsumes these items into a common list.

However, categories of data which are derived from another category will often use category keys which refer to the same quantities. In these cases, the keys are not implicitly equivalent (as would be the case if the categories were hierarchically related) but they are “linked” using the DDL attribute _name.parent_item_id. Here is the definition of an item in the category GEOM which is linked to a category key in the category POSITION (see Example 1).

```
  _definition.id             '_geom.vertex1_id'
  _name.category_id            geom
  _name.object_id              vertex1_id
  _name.linked_item_id       '_position.object_id'
  _type.container              Single
  _type.contents               Uchar
```

The _name.linked_item_id attribute specify that _geom.vertex1_id has a value that is common to one of the unique values of the item _position.object_id. This linkage implies that _position.object_id is a "key" unique item in the category POSITION. The same relationships also apply for the items _geom.vertex2_id and _geom.vertex3_id, which are shown below in an example data list.
As in §5.4.2, specific values in this list can be accessed via their unique extension names. However, because of the defined relationship between the vertex ID's and the _position.object_id (in Example 1), these can be used to “point” to specific packets and items in the POSITION category using the <category>[<key>].<extension> construction. The With command is described in the next section.

```python
With p as position
  Loop g as geom {
    If (g.type == "point") {
      PointList += Tuple([g.vertex1_id, p[g.vertex1_id].vector_xyz])
    }
    Else if (g.type == "line") {
      LineList += Tuple([Tuple([g.vertex1_id, g.vertex2_id],
                             Tuple([p[g.vertex1_id].vector_xyz,
                                   p[g.vertex2_id].vector_xyz])])
    }
  }
```

This illustrates how values from the category list can be directly accessed simply by appending the name extensions to the item which is linked to the key of that list. The value of LineList[0] is `[["origin","body-diagonal"],[[0.0,0.0,0.0],[5.0,5.0,5.0]]]

### 5.5 WITH STATEMENT

The With statement is identical to the Loop statement except that the list pointer is not incremented. This statement is used only to identify the current list object within scope and context as a local object. The general form is as follows.

```python
With local as list { *expression block* }
```

This statement is very useful for accessing data items in the current packet of a category lists. This enables items in a list to be addressed as name extension attributes, just as in Loop.

```python
With p as atom_site
  If (label == p.id)  x = p.frac_vector
```

Note the braces about the expression block are required for multiline expressions.

### 5.6 WHERE STATEMENT

The Where operator is used to test all elements in arrays or lists, which may be of indeterminate length. This operator has the general form:

```python
Where (expr) { "expression block" }
Else    { "expression block" }
```

If A and B are arrays of the same shape then the statement works element by element.

```python
Where (A>0) { B = 1.0/A }
Else        { B = large }
```
It is difficult to write an equivalent statement to this using other operators because the shape of arrays (e.g. the number of dimensions) might be unknown.

5.7 **BREAK TERMINATOR**

Repetitive blocks can be exited prematurely with the `Break` keyword. The general form of the statement is as follows.

**Break**

For example, in the sequence

```plaintext
Do i=1:10 {
  Do j=i+1:10 {
    If (a[i] < a[j]) Break
  }
}
```

5.8 **NEXT TERMINATOR**

Repetitive blocks can be reset prematurely with the `Next` keyword. The general form of the statement is as follows.

**Next**

For example, in the sequence

```plaintext
Do i=1:10 {
  Do j=i+1:10 {
    If (a[i] < a[j]) Next
  }
}
```

5.9 **IF/ELSEIF/ELSE STATEMENTS**

The standard `If/ElseIf/Else` statements have the following form and sequence. The `If` statement must precede all others in the sequence. The `Else` statement must, if used, follow all others. There may be any number of `ElseIf` statements.

```plaintext
If (expr) { *expression block* }
Else If (expr) { *expression block* }
Else { *expression block* }
```

Braces around the expression blocks are necessary if they contain more than one statement.

5.10 **SWITCH/CASE/DEFAULT STATEMENTS**

The `Switch` statements are used to execute expression blocks according to a match with an enumerated value. The operators have the general form:

```plaintext
Switch (var) {
  Case (val1,.., valN) { *expression block* } # case
  Case (valM,.., valQ) { *expression block* } # case
  Default { *expression block* } # default
}
```

where `var` is the variable NAMED object whose value is tested against values `val1,.., valQ`. When there is a match, the corresponding expression block is entered. NOTE that all case lists are tested and more than one expression block may be entered. If no case blocks are entered, the default block is entered.
Here is an example of a `Switch` sequence of statements.

```
Switch (NUM) {
    Case (5) { .... }
    Case (7,8,6) { ..... }
    Case (1:4) {...... }
    Default { .... }  }
```

The case labels must be constant expressions.

### 6. INTRINSIC FUNCTIONS

*dREL* has an extensive set of intrinsic functions, which are listed in this section according to the following classes.

- **CONVERSION and MANIPULATION**
- **TRIGONOMETRIC**
- **MATHEMATICAL**
- **DISCIPLINE**

#### 6.1 CONVERSION AND MANIPULATION Functions.

These functions are responsible for fixing the TYPE of the contained object.

- **`Complex()`** Convert two arguments (Real, Imag) into a Complex number
- **`Real(), Imag()`** Returns real and imaginary part of Complex argument
- **`Integer()`** Convert argument into an integer number
- **`Float(), Rem()`** Convert to real number, get remainder of real number
- **`Int(), Nint()`** Convert to truncated integer, rounded-up integer value
- **`List()`** Convert arguments into a List object.
- **`Tuple()`** Convert arguments into a Tuple object.
- **`Table()`** Convert arguments into a Table object.
- **`Array()`** Convert arguments into an Array object.
- **`Numb()`** Convert the character argument into the ascii number equivalent.
- **`Char()`** Convert the ascii number argument into a character equivalent.
- **`Minor()`** Generate a matrix of minor elements from the matrix argument.
- **`Cofactor()`** Generate a matrix of cofactor elements from the matrix argument.
- **`Adjoint()`** Generate a matrix of adjoint elements from the matrix argument.
- **`Inverse()`** Generate a matrix of inverse elements from the matrix argument.
- **`Transpose()`** Generate a matrix of transposed elements from the matrix argument.
- **`Eigen()`** Get eigenvalues and vectors of a 3x3 matrix and return as three tuples containing four elements (value plus vector of direction cosines).

#### 6.2 TRIGONOMETRIC Functions.

These functions are responsible for performing trigonometric operations on the argument.

- **`Sin(), Cos(), Tan()`** Sine, cosine and tangent functions of radian arguments.
- **`Sind(), Cosd(), Tand()`** Sine, cosine and tangent functions of degree arguments.
- **`Asin(), Acos(), Atan()`** Arcsine, cosine and tangent functions as radians.
- **`Arcsin(), Arccos(), Arctan()`** Arcsine, cosine and tangent functions as radians.
- **`Asind(), Acosd(), Atand()`** Arcsine, cosine and tangent functions as degrees.
- **`Atan2(a,b), Atan2d(a,b)`** Arctangent function in radians and degrees
- **`Phase()`** Get the phase in radians for a Complex number.
Exp(), ExpIm(), ExpImag()  Exponential functions with Real and Complex arguments.
Log(), Ln()  Base-10 and natural logarithm functions.
Pi, TwoPi  Values of π and 2π.

6.3 MATHEMATICAL FUNCTIONS

These functions are responsible for performing mathematical operations on the arguments.

Sqrt()  Get square root of number.
Mod()  Modulus of arg1 to base arg2.
Abs(), Magn()  Absolute value of the argument.
Sign()  Sign of argument 2 applied to argument 1.
Sum()  Sum all of all the values in the list object.
First(), Last()  Get the first and last element of a list or character string.
Strip(list, n)  Strip the nth element from the list. (n=0,1,2...)
Len()  Get the length of a list or character string.
Map(list, func)  Apply the function func to each element in the list.
Sort()  Sort all elements in a list from small to large.
Sort(list, func)  Sort the list according to the function func.
Reverse()  Reverse the order of a list.
TopLo(), TopHi()  Sort all elements in a list from small to large; large to small.
Dim()  Return an integer list of dimension lengths. Zero value is end of array.
Det()  Get the determinant of a matrix
Dot(), Cross()  Scalar and vector product of two vectors.
Norm()  Root mean square value of elements in a list or vector.
MaxI(list, ind)  Maximum value in list. Index of max value returned as argument 2.
MinI(list, ind)  Minimum value in list. Index of min value returned as argument 2.
Max(), Min()  Maximum and minimum values in the list.
SubString(s1, s2)  Returns TRUE if string s1 is a substring of s2.
Eigen(mat)  Return sorted list of three (value, vector) tuples.

6.3 DISCIPLINE FUNCTIONS

Specific functions may be defined in a data dictionary using the a definition save frame and DDL attributes. These frames are opened with "save_function.<FunctionName>". The typing of the function value is specified using the TYPE attributes. The definition of the a discipline function within the method expression is achieved as follows:

```
Function <FunctionName>(
  <arg1>: [ <ContainerType>, <ContentsType> ],
  <arg2>: [ <ContainerType>, <ContentsType> ], etc. )
  { <expression evaluating FunctionName in terms of the input arguments> }
```

Note that an argument may be a container type "Category" and contents type "Tag".

In the Crystallographic CORE dictionary the following functions are already defined.

AtomType(label)  Extract the “atom_type” element symbol from an atom label string label.
Closest(v, u)  Returns \([w, t]\) where \(w\) is the closest real space vector transformation of \(v\) to \(u\), and \(t\) is the integer cell vector that converts \(v\) to \(w\).
SeitzFromJones(text)  Converts a Jones-Faithful equiv. pos. text \((x,y,z)\) into a 4x4 Seitz matrix.
SymEquiv(s, cat, v)  Converts a coordinate vector \(v\) into a vector transformed by the symmetry seitz matrix extracted from category \(cat\) using index \(n\) from symop code \(s\).
SymLat(s) Convert the symop code \( n_{jkl} \) into a lattice vector \([j-5, k-5, l-5]\)

SymNum(s) Convert the symop code \( n_{jkl} \) into a symmetry integer \( n \). \( (n=0,1,2...) \)

Symop(index, lvect) Convert symmetry equivalent position number \( index \) and cell lattice vector \( lvect \) to the symop code \( n_{jkl} \). \( (n=1,2,3...) \)

7. LIST OPERATORS

7.1 STRING CONCATENATION

The following properties of strings apply.

- Concatenation of ASCII and UNICODE strings results in a UNICODE string.
- Character strings are immutable.
- There is no "char" type. Strings of length 1 are used.

7.1.1 Concatenation of literals

Multiple sequential string literals will be concatenated automatically in statements. E.g.

\[
x = "string literals that are adjacent" " are concatenated"
\]

equivalent to

\[
x = "string literals that are adjacent are concatenated"
\]

7.1.2 Concatenation of objects

The operators + and * may be applied to string objects. Here is an example of the + operator.

\[
s1 = "this" ; s2 = " and that"
s3 = s1 + s2
\]

The object \( s3 \) now holds “this and that”.

Strings made up of multiple instances of the same character sequence can be generated by the * operator, as below.

\[
s4 = "-"*10
\]

The object \( s4 \) now holds a string "----------". The * operator can be applied to named objects as well.

\[
s4 = "-EOF-" ; s5 = s4*3
\]

The object \( s4 \) now holds a string "-EOF--EOF--EOF-".

7.2 LIST MEMBERSHIP

It is possible to test objects containing lists of strings for the “membership” of specific strings. These tests are equivalent to looping through the lists and applying the standard string equivalence operators “==” and “!”=”, as illustrated in the following example statements.

\[
cnt = List(["data_", "global_", "save_", "stop_", "loop_"])\]

Do i=0,4 { If("stop_" == cnt[i]) Break ;}

The last statement is problematical because the length of the list of items being tested needs to be known. It may be replaced simply by:

\[
If ("stop_" in cnt) { ... }
\]
This works only if elements of the container are of the same type. The negation test for membership of a list also applies. E.g.

```
If ("cell_" not in cnt) { ... }
```

### 7.3 List Notation

The following notation is available for the formation of lists from existing named lists.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new = list[i]</td>
<td>New copy of entire list.</td>
</tr>
<tr>
<td>new = list[n:m:i]</td>
<td>New list with elements from indices n to m in steps of i.</td>
</tr>
<tr>
<td>new = list[first+1:last-1]</td>
<td>New list without the first and last elements. #not implemented</td>
</tr>
<tr>
<td>val = list[1]</td>
<td>Extract a single value.</td>
</tr>
<tr>
<td>new = list1 + list2</td>
<td>New list of list1 concatenated with list2.</td>
</tr>
<tr>
<td>list1 += val</td>
<td>Append val to list1.</td>
</tr>
<tr>
<td>list[i:j] = list2</td>
<td>Cut and paste ALL of list2 into the elements i to j.</td>
</tr>
<tr>
<td>new = list!n</td>
<td>New list composed of n copies of list.</td>
</tr>
<tr>
<td>new = n*list</td>
<td>New list with list elements multiplied by a number n.</td>
</tr>
<tr>
<td>vec = vec1 + vec2</td>
<td>Vector (cross) product. ((-4,8,-4) = \text{Vector} ([4,5,6]) \times \text{Vector} ([8,9,10]))</td>
</tr>
<tr>
<td>vec = mat * vec1</td>
<td>Post-matrix vector multiply. E.g. ([32,77,112] = \text{Matrix} ([1,2,3],[4,5,6],[7,8,9]) \times \text{Vector} ([4,5,6]))</td>
</tr>
<tr>
<td>vec = vec1 * mat</td>
<td>Pre-matrix vector multiply. E.g. ([66,81,96] = \text{Vector} ([4,5,6]) \times \text{Matrix} ([1,2,3],[4,5,6],[7,8,9]))</td>
</tr>
<tr>
<td>list = list + x</td>
<td>Add value x to all elements of an existing list.</td>
</tr>
</tbody>
</table>

### 7.4 Array Notation

The following notation applies strictly to Array objects.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var = mat[n,m]</td>
<td>Variable contains the value of the matrix element (n,m).</td>
</tr>
<tr>
<td>mat[p,q] = x</td>
<td>Matrix element (p,q) is replace with the value of x. #Its immutable</td>
</tr>
<tr>
<td>vec = mat[:j]</td>
<td>Vector formed from jth column of row matrix elements. # mat.v[:j]</td>
</tr>
<tr>
<td>vec = mat[first:last-1,k]</td>
<td>Vector formed from kth column of row elements first to last-1.</td>
</tr>
<tr>
<td>vec = vec1 + val</td>
<td>Scalar addition. ([9,10,11] = \text{Vector} ([4,5,6]) + 5)</td>
</tr>
<tr>
<td>vec = Function(vec1)</td>
<td>Vector function. ([1,2,0] = \text{Mod} ([4,5,6], 3)) for ((\text{Mod}, \text{Int}),)</td>
</tr>
<tr>
<td>vec = vec1 + vec2</td>
<td>Vector addition. ([12,14,16] = \text{Vector} ([4,5,6]) + \text{Vector} ([8,9,10]))</td>
</tr>
<tr>
<td>var = vec1 * vec2</td>
<td>Scalar (dot) product. (8<em>4+9</em>5+10*6 = \text{Vector} ([4,5,6]) \times \text{Vector} ([8,9,10]))</td>
</tr>
<tr>
<td>vec = vec1 ^ vec1</td>
<td>Vector (cross) product. ((-4,8,-4) = \text{Vector} ([4,5,6]) \times \text{Vector} ([8,9,10]))</td>
</tr>
<tr>
<td>vec = mat * vec1</td>
<td>Post-matrix vector multiply. E.g. ([32,77,112] = \text{Matrix} ([1,2,3],[4,5,6],[7,8,9]) \times \text{Vector} ([4,5,6]))</td>
</tr>
<tr>
<td>vec = vec1 * mat</td>
<td>Pre-matrix vector multiply. E.g. ([66,81,96] = \text{Vector} ([4,5,6]) \times \text{Matrix} ([1,2,3],[4,5,6],[7,8,9]))</td>
</tr>
<tr>
<td>mat = mat1 * mat2</td>
<td>Matrix multiply. Matrices must have concordant shapes.</td>
</tr>
</tbody>
</table>