dREL Language Specifications

Appendix to the paper:
dREL: A RELATIONAL EXPRESSION LANGUAGE FOR DICTIONARY METHODS

Nick Spadaccini, Ian R. Castleden, Doug du Boulay, Sydney R. Hall

School of Biomedical, Biomolecular and Chemical Sciences, The University of Western Australia, Nedlands 6009, Australia

Email: *Nick.Spadaccini@uwa.edu.au, Sydney.Hall@uwa.edu.au*

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`).

1. CHARACTER STRINGS

1.1 Dictionary definition

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

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.

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"'
```

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\""
```

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\nare\nmulti line\n"
```

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. *INTEGER* NUMBERS

dREL supports decimal 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.

2.1 Dictionary definition

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

2.2 Inline definition

2.2.1 Decimal integers

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

2.2.2 Hexadecimal integers

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

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.

3.1 Dictionary definition

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

3.2 Inline definition

3.2.1 Decimal real numbers

The syntax of a decimal real number is: `[+-]?([0-9]+.[0-9]*|.[0-9]+)[[Ee][+-]?[0-9]+]?`
An example decimal real number is: `-7893.8221` or `-7.89382e+3`

3.3 Explicit definition

Conversion to real number is achieved with the function:

- *Float())*

4. *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.

4.1 Dictionary definition

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

4.2 Explicit definition

Conversion to a complex number is achieved with the function:

- *Complex (Nreal, Nimag)*

5. *MEASURED* NUMBERS

A *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. *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.

5.1 Dictionary definition

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

```
_type.contents                 Real
_type.purpose                  Measured
```

The value of the attribute `_type.contents` can also be *Integer* or *Complex*.

5.2 Inline definition

5.2.1 Measured numbers

The syntax of a measurement number is: `[Real|DecimalInteger]\([1-9][0-9]*\)`
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.

5.3 Explicit definition

Conversion to a measurement number is achieved with the function:

- *Measure (val, su)*

Container Types for dREL

dREL supports the container types

- List *List data is bounded by square brackets* [ ]
- Array *Array data is bounded by square brackets* [ ]
- Matrix *Matrix data is bounded by square brackets* [ ]
- Table *Table data is bounded by curly brackets* { }
- Single

dREL also supports the nesting and mixing of container types i.e. the definition

```
_type.container                List
```

```
_type.contents                  Real
-type.dimension                 (5(3))
```

refers to a list of five arrays; each array contains three real numbers.

1. *LIST* CONTAINERS

List containers are objects with the following properties.

- **Type**: contained items may be of any, but the same, TYPE.
- **Dimension**: Lists are single dimensioned.
- **Size**: the length of a list need *not* pre-defined.
- **Access**: indexed by integers *(implied starting index is 0).*
- **Shape**: bounded by [....] and 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.

1.1 Dictionary definition

The dictionary definitions declare the nature of a *List* container with attribute declarations. Here are such declarations for a list of real numbers of nine elements.

```
_type.container                List
_type.contents                 Integer
```

1.2 Inline definition

Lists may be defined inline using the *List*(…) function. E.g.

`List([ 1, 7, 3, 10 ])` *which is also implied by* [1,7,3,10]

2. *ARRAY* CONTAINERS

Array containers are objects with the following properties.

- **Type**: may contain items of *any* TYPE.
- **Dimension**: are single or multi dimensioned.
- **Size**: predetermined upper extents; minimum elements assumed to be 1.
- **Access**: indexed by integers *(implied starting index is 0).*
- **Shape**: bounded by [....] and may be nested.

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

2.1 Dictionary definition

The dictionary definitions declare the nature of a *Array* container with attribute declarations. Here are the attributes for defining an array of binary numbers.

```
_type.container                Array
_type.contents                 Binary
```

2.2 Inline definition

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

`Array([[126,255,0],[123,245,10]])` *which is also implied by* [[126,255,0],[123,245,10]] *for a data item defined as* `_type.container  Array`

3. *MATRIX* CONTAINERS

Matrix containers are objects with the following properties.

- **Type**: only contain items of *number* TYPE.
- **Dimension**: are single or multi dimensioned.
- **Size**: predetermined upper extents; minimum elements assumed to be 1.
- **Access**: indexed by integers *(implied starting index is 0).*
- **Shape**: bounded by [....] and may be nested.

Matrices 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 Dictionary definition

The dictionary definitions declare the nature of a *Matrix* container with attribute declarations. Here are the attributes for defining a vector of three real numbers, indexed from 0 to 2.

```
_type.container              Matrix
_type.contents               Real
_type.dimension               [3]
```

3.2 Inline definition

Vectors may be defined inline using the *Matrix*(….) function. E.g.

`Matrix([10.2, 12.3, 7.4])` *which is also implied by* [10.2,12.3,7.4] *for a data item defined as* `_type.container  Matrix`

4. *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.
- **Shape**: bounded by {...} and may be nested.

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.

4.1 Dictionary definition

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

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

A *Table* differs from a *List* 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.

4.2 Explicit definition

Conversion of a sequence of objects to a new list is achieved with the function *Table ('key':val,..). E.g.*

`Table("left":"links","right":"recht")` *implied by* `{"left":"links","right":"recht"}`

5. *SINGLE* CONTAINERS

Single containers are <u>a single value</u> 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.

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.

```
_type.container              Single
_type.contents               Real
```

5.2 Inline definition

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

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

Language Basics

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

1. Assignment Expressions

1.1 Named objects

A NAMED object or "variable" in dREL may only be created on assignment. The *typing* of a variable is by coersion. The *scope* of a variable is *local*.

1.2 Assignment statements

1.2.1 The process of <u>object transfer</u> is initiated with the "=" character which transfers the value of the right-hand expression of objects *Robjects* to the left-hand objects *Lobjects*. The general form of the object transfer is:

Lobjects = *Robjects* or *Lobjects* = { *multi-line expression* }

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
```

Robjects 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
```

1.2.2 The process of <u>object incrementation</u> is initiated with the "+=" digraph which increments the values of the right-hand expression of objects *Robjects* to the left-hand objects *Lobjects*. The general form of an object incrementation is:

Lobjects += *Robjects*

In the example below the value of the *literal* Integer "`1`", is added to the existing value in a mutable NAME object, the single variable "`x`".

```
x += 2
```
i.e. if the value of x is initially 5, becomes 7

Lobjects may also be a muli-element container whereas *Robjects* may be either a single value or a multi-element container E.g.

`vect += 1`	*i.e. if vect is initially [3,3,3], becomes [4,4,4]*
`vect += [1,2,3]`	*i.e. if vect is initially [3,3,3], becomes [4,5,6]*

1.2.2 The process of <u>object decrementation</u> is initiated with the "`-=`" digraph which decrements the values of the right-hand expression of objects *Robjects* to the left-hand objects *Lobjects*. The general form of an object incrementation is:

Lobjects -= *Robjects*

In the example below the value of the *literal* Integer "`1`", is subtracted from the existing value in a mutable NAME object, the single variable "`x`".

```
x -= 2
```
i.e. if the value of x is initially 5, becomes 3

Lobjects may also be a muli-element container whereas *Robjects* may be either a single value or a multi-element container E.g.

`vect -= 1`	*i.e. if vect is initially [3,3,3], becomes [2,2,2]*
`vect -= [1,2,3]`	*i.e. if vect is initially [3,3,3], becomes [2,1,0]*

1.2.3 The process of <u>object appending</u> is initiated with the "`++=`" trigraph which appends the values of the right-hand expression of objects *Robjects* to the <u>end of left-hand objects</u> *Lobjects*. The general form of an object appending is:

Lobjects ++= *Robjects*

Lobjects <u>must be</u> a muli-element container whereas *Robjects* may be either a single value or a multi-element container E.g.

`vect ++= 1`	*i.e. if vect is initially [3,3,3], becomes [3,3,3,1]*
`vec2 ++= [1,2,3]`	*i.e. if vec2 is initially [3,3,3], becomes [3,3,3,1,2,3]*
`matx ++= [[1,2,3]]`	*i.e. if matx is initially [[3,2,1]], becomes [[3,2,1],[1,2,3]]*

1.3 Assignment TYPING

In dREL object types are not declared. We have already seen that the typing of *Robjects* 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 *Robjects* 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.

2. TYPE COERCION RULES

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

Integer → Real → Complex

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. 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* and *Table* 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.

```
a = 234 ;  y = 45 ;  z = -2
b = (y + z)/2.0
```

```
c = (45 + 72 *
       (93 + 4) + z)
```

5. SUPPORTED ESCAPE SEQUENCES

The following special 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*
`\x`	*hexadecimal bit pattern*
`\0`	*null character*
`\\`	*backslash (\)*

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:

- Do
- Repeat
- For
- Loop
- With
- Break
- Next
- If/ElseIf/Else

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.

1. *DO* STATEMENT

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

Do *index* = *first*, *last*, *incr* { *expression block* }

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]; }
```

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;….. }
```

3. *FOR* STATEMENT

Manipulation of List items is provided with 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.

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

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.

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.keyid             '_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

_definition.id             '_position.vector_xyz'
_name.category_id           position
_name.object_id             vector_xyz
_type.container             Matrix
_type.contents              Real
_type.dimension             [3]
```

In a data file these items might appear in a looped list (abbreviated) as follows.

```
loop_
    _position.number
    _position.object_id
    _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
}
```

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 <u>derived</u>, 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.

*A*ll 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 that 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               Text
```

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.

```
loop_
    _geom.type
    _geom.vertex1_id
    _geom.vertex2_id
    _geom.vertex3_id
        point      origin          .                   .
        line       origin          body-diagonal       .
        line       body-diagonal   diagonal-terminal   .
        triangle   origin          body-diagonal       diagonal-terminal
```

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 used the example dREL script below is described in the next section and the list-append operator "`++=`" is described below.

```
With p as position
Loop g as geom {
        If (g.type == "point") {
              PointList ++= Tuple(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. Executing this script results in the following values strings:

`PointList[0]` is `("origin",[0.,0.,0.])`
`LineList[0]` is `("origin","body-diagonal"),([0.,0.,0.],[5.,5.,5.])`
`LineList[1]` is `("body-diagonal","diagonal-terminal"),([5.,5.,5.],[10.,10.,10.])`

5. *WITH* STATEMENT

The *With* statement is identical to the *Loop* statement except that the list pointer is <u>not</u> 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.

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*.

```
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.

6. *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

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

7. *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

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

8. *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.

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.

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

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 trucated integer, rounded-up integer value
List()	Convert arguments into a *List* object.
Table()	Convert arguments into a *Table* 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 lists containing four elements (value plus vector of direction cosines).

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 p and 2p

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 argment 2 applied to argument 1.
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.
Sort()	Sort all elements in a list from small to large.
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 max 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) Lists.

4. DISCIPLINE-SPECIFIC 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...)

List Operators

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.*

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"
```

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-"`.

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)  { … }
```

3. *LIST* NOTATION

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

new = *list*[**:**]	New copy of entire list.
new = *list*[**n:m:i**]	New list with elements from indices n to m in steps of i.
new = *list*[**n:m**]	New list of elements from indices n to m in steps of 1.
val = *list*[**1**]	*val* becomes the value of the second element of *list*.
new = *list1* + *list2*	New list of *list1* concatenated with *list2*.
new = **[***list1*, *list2***]**	New list of *list1* concatenated with *list2*.
val1 += *val2*	Increment *val1* with *val2*.
list1 += *val*	Increment all elements in *list1* with *val*.
list1 += *list2*	Increment matching elements in *list1* with values in *list2*.
list1 ++= *val*	Append *val* to *list1*.
list1 ++= *list2*	Append *list2* to *list1*.
val1 -= *val2*	Decrement *val1* with *val2*.
list1 -= *val*	Decrement all elements in *list1* with *val*.
list1 -= *list2*	Decrement matching elements in *list1* with values in *list2*.
list[**i:j**] = *list2*	Cut and paste ALL of list2 into the elements i to j-1.

4. *ARRAY* NOTATION

The following notation applies strictly to *Array* objects.

var = *mat*[**n,m**]	Variable contains the value of the matrix element (n,m)
vec = *vec1* **+** *val*	Scalar addition. [9,10,11] = Vector([4,5,6]) + 5
vec = *Function*(*vec1***)**	Vector function. [1,2,0] = *Mod*([4,5,6], 3) for (*Mod*, *Int*,)
vec = *vec1* **+** *vec2*	Vector addition. [12,14,16] = Vector([4,5,6]) + Vector([8,9,10])
var = *vec1* ***** *vec2*	Scalar (dot) product. 8*4+9*5+10*6 =Vector([4,5,6])*Vector([8,9,10])
vec = *vec1* **^** *vec2*	Vector (cross) product. (-4,8,-4) = Vector([4,5,6]) ^ Vector([8,9,10])
vec = *mat* * *vec1*	Post-matrix vector multiply. E.g. [32,77,112] = Matrix([[1,2,3],[4,5,6],[7,8,9]]) * Vector([4,5,6])
vec = *vec1* * *mat*	Pre-matrix vector multiply. E.g. [66,81,96] = Vector([4,5,6]) * Matrix([[1,2,3],[4,5,6],[7,8,9]])
mat = *mat1* * *mat2*	Matrix multiply. Matrices **must** have concordant shapes.