International Tables for Crystallography (2006). Vol. G, Chapter 5.2, pp. 488–498.

5.2. STAR File utilities

BY N. SPADACCINI, S. R. HALL AND B. MCMAHON

5.2.1. Introduction

The STAR File, described in Chapter 2.1, has a simple format intended to allow the flexible and extensible representation of data without regard to specific data models. In crystallography and related disciplines, the restricted format chosen for the Crystallographic Information File (CIF, Chapter 2.2) and Crystallographic Binary File (CBF, Chapter 2.3) lends itself to rather flat data models. In particular, the relationships between data items enforced through DDL2 dictionaries in applications such as mmCIF (Chapter 3.6) are essentially equivalent to the data structures and relationships of a relational database. Of course, properly normalized relational tables can represent a hierarchy of structure, although this may not be an efficient representation.

There are other applications, such as the molecular information file (MIF, Chapter 2.4), that make use of additional features of the STAR File, such as multiple-level loop structures, global variable scoping and data-instance encapsulation in save frames. These applications may more efficiently represent certain hierarchical or object-oriented data models.

While particular applications require software tools tailored to their specific purposes, it is helpful to have programs or libraries capable of manipulating arbitrary STAR File data, relying solely on the syntax rules and format of the STAR File and taking no account of the semantic content of the included data.

In this chapter, the stand-alone program *Star_Base* is described in detail. This program uses a local query language to demonstrate the ability to retrieve or re-order data with their associated context. There is also a brief review of *Star.vim* and *StarMarkUp*, applications for editing and browsing STAR Files. The chapter concludes by reviewing a number of object classes and libraries for a variety of STAR and generalized CIF applications: prototypical approaches *OOSTAR* and *CIF++*, *CIFOBJ* and *starlib* used by major macromolecular data repositories, and the document-object model package *StarDOM*.

5.2.2. Data instances and context

In a STAR File, a data item consists of a *value*, which is a simple ASCII character string, and an associated identifier or *data name* which precedes the value, and is invariably an ASCII character string beginning with an underscore character and not including any white-space character, such as `_date` or `_chemical_formula_sum`. (The detailed and formal syntax rules for STAR Files are given in Chapter 2.1.)

5.2.2.1. Single and multiple values

A data item may have a single value, in which case the data name may immediately precede the data value, separated only by white space, *e.g.*

```
_chapter_title  'STAR File utilities'
```

Alternatively, a data item may occur multiple times, in a vector or a list. In such a case, the data identifiers appear in a loop header and the values follow in the order of presentation in the loop header. For the simple example of a tabular array, the loop header plays the role of column header, *e.g.*

```
loop_
   _chapter_number
   _chapter_title
       5.2  'STAR File utilities'
       5.3  'Syntactic utilities for CIF'
```

Here the instances of the data item identified by the data name `_chapter_number` have two values, `5.2` and `5.3`. Likewise the instances of the data item identified by `_chapter_title` have two values.

Note an important point: the example has been chosen to suggest to the reader a tabular relationship between the two data items, and in many STAR File applications such a relationship is intended and perhaps formalized through an external dictionary defining the relationships between these data names. However, *the existence of such a relationship is not mandated by the STAR File syntax*. It is legitimate for a generic STAR application to extract a single data item from such an aggregated loop without making any supposition about its relationship with other data items in the same loop. (It should be emphasized that in practice such physical juxtaposition of data items will almost invariably represent a real relationship, and that most application-specific programming will depend on this fact; but it is not an essential component of STAR in its most abstract form.)

It is also axiomatic that the ordering of the multiple values within a list structure has no intrinsic significance in the STAR paradigm. (Again, specific applications may override this by enforcing an ordering, but this is not fundamental to STAR.)

5.2.2.2. Loop packets and context within lists

Where multiple data names are declared in a loop header, STAR does however enforce the notion of a 'loop packet'. The loop packet is the data structure including all individual data values at a particular iteration through the loop. Hence, in the simple example above, `5.2` and `STAR File utilities` comprise the tuple of values in a single loop packet. For the single level of loop considered so far, the loop packet plays the role of a table row.

For nested loops, the situation is more complex. Consider Fig. 5.2.2.1, which is an example of quantum chemistry basis sets for hydrogen and lithium. (The examples in this chapter are derived from various test applications, and do not represent specific adopted exchange protocols in the selected subject areas.) For each element, a list of basis sets is presented, each containing a set of parameters and a table of functional values. At the outermost level of looping in this example, a loop packet comprises all the data associated with an individual atom type, for example hydrogen. At the next inner level of looping, a loop packet corresponds to an individual basis set (including its embedded table of

Affiliations: N. SPADACCINI, School of Computer Science and Software Engineering, University of Western Australia, 35 Stirling Highway, Crawley, Perth, WA 6009, Australia; SYDNEY R. HALL, School of Biomedical and Chemical Sciences, University of Western Australia, Crawley, Perth, WA 6009, Australia; BRIAN MCMAHON, International Union of Crystallography, 5 Abbey Square, Chester CH1 2HU, England.

```
data_Gaussian

loop_
     _basis_set_atomic_name
     _basis_set_atomic_symbol
     _basis_set_atomic_number
     _basis_set_atomic_mass
    loop_
     _basis_set_contraction_scheme
     _basis_set_funct_per_contraction
     _basis_set_primary_reference
     _basis_set_source_exponent
     _basis_set_source_coefficient
     _basis_set_atomic_energy
        loop_
     _basis_set_function_exponent
     _basis_set_function_coefficient

      hydrogen    H    1    1.0079
#     --------
 (2)->[2]   1:    PKC1.1.1   R44 .       -0.485813
      1.3324838E+01    1.0
      2.0152720E-01    1.0 stop_
 (2)->[2]   1:    PKC1.2.1   R33 .       -0.485813
      1.3326990E+01    1.0
      2.0154600E-01    1.0 stop_
 (2)->[1]   2     PKC1.14.1  R24 R24     -0.485813
      1.3324800E-01    2.7440850E-01
      2.0152870E-01    8.2122540E-01 stop_
 (3)->[2]   2:1   PKC1.23.1  R75 R75     -0.496979
      4.5018000E+00    1.5628500E-01
      6.8144400E-01    9.0469100E-01
      1.5139800E-01    1.0000000E+01 stop_ stop_

     lithium    Li   3     6.94
#    -------
 (4)->[4]   1:    PKC3.1.1   R44 .       -7.376895
       3.4856175E+01     1.0
       5.1764114E+00     1.0
       1.0514394E+00     1.0
       4.7192775E-02     1.0 stop_
 (9,4)->[3,2]  7:2:1,3:1
                  PKC3.9.1   R2  R98     -7.431735
       921.271    0.001367       138.730    0.010425
       31.9415    0.049859       9.35329    0.160701
       3.15789    0.344604       1.15685    0.425197
       0.44462    0.169468       0.44462   -0.222311
       0.076663   1.116477       0.028643   1.0
       1.488      0.038770       0.2667     0.236257
       0.07201    0.830448       0.02370    1.0     stop_
 (4,3)->[3,2]   4:2:1,2:1
                  PKC3.30.1  R77 R77     -7.419509
       1.09353E+02    1.90277E-02 1.64228E+01    1.30276E-01
       3.59415E+00    4.39082E-01 9.05297E-01    5.57314E-01

       5.40205E-01   -2.63127E-01 1.02255E-01    1.14339E+00
       2.85645E-02    1.00000E+00

       5.40205E-01    1.61546E-01 1.02255E-01    9.15663E-01
       2.85645E-02    1.00000E+00  stop_ stop_
```

Fig. 5.2.2.1. Example quantum chemistry basis set functions in STAR File format.

coefficients). At the innermost loop level, a loop packet is simply a row within a table of exponents and coefficients of the basis set function.

If one were to treat this example file as a database of indeterminate structure and query the values associated with one of the data names, for example, `_basis_set_function_exponent`, one would retrieve a series of strings `1.3324838E+01`, `2.0152720E-01` *etc.* However, the value strings in themselves are insufficient to allow the reconstitution of any data structure in the file. One also needs an expression of the levels within the nested loop structure at which the values were located, and an indication that they were associated with different packets of information at those various levels. This additional information about the context of each value is sufficient to determine its position within the data structure

```
data_Gaussian
loop_
    loop_
        loop_
     _basis_set_function_exponent
        stop_
    stop_

        1.3324838E+01 2.0152720E-01 stop_
        1.3326990E+01 2.0154600E-01 stop_
        1.3324800E-01 2.0152870E-01 stop_
        4.5018000E+00 6.8144400E-01 1.5139800E-01 stop_
    stop_

        3.4856175E+01 5.1764114E+00 1.0514394E+00
        4.7192775E-02 stop_

        921.271 138.730 31.9415 9.35329 3.15789 1.15685
        0.44462 0.44462 0.076663 0.028643 1.488 0.2667
        0.07201 0.02370 stop_

        1.09353E+02 1.64228E+01 3.59415E+00 9.05297E-01
        5.40205E-01 1.02255E-01 2.85645E-02 5.40205E-01
        1.02255E-01 2.85645E-02 stop_
    stop_
```

Fig. 5.2.2.2. Retrieval from the example file in Fig. 5.2.2.1 of the value of `_basis_set_function_exponent` with associated context.

without any other *a priori* information regarding the data model. The context is most easily expressed by listing the output values in STAR File format.

Fig. 5.2.2.2 is an output listing of the requested values for this example, where the context is expressed as the innermost of three nested loop levels and distinct packets at this level are indicated. It will be seen also that by tracing the disposition of `stop_` words the embedding within higher-level loop packets can also be inferred.

5.2.2.3. Context in data sets

Another indicator of context in the previous example is the data-block header, which was reproduced in the output of Fig. 5.2.2.2.

The STAR File allows data instances in three types of location: in a data block, in a save frame or in a global block.

The usual way to partition a STAR File is by data blocks; each such block represents a data set in which a data name (associated with a single or multiple values) may be declared once only.

Data blocks may include save frames. A save frame is an encapsulated subsidiary data set, effectively insulated from the contents of the surrounding data block, in which data items may occur that have the same names as items in the parent data block. Indeed, 'parent' is potentially a misleading term, since no relationship is implied between the data within a save frame and those in the data block in which the save frame occurs. A reference to a save frame may, however, occur as a data *value* within the data block where the save frame is specified. Recall from Section 2.1.3.6 that save frames within a data block are uniquely identified by the *frame-code* header.

Global blocks may also occur in a STAR File, preceding or interspersed between data blocks. For each data item defined within a global block, that definition is inherited by each succeeding data block that does not contain an internal definition of a data item with the same name. If there is a definition of a data item with the same name within a data block, that internal definition overrides the global definition within that data block. The situation is then re-evaluated in the next data block. If that data block does not contain an internal definition, the global definition holds.

```
data_reaction

save_methyl
  loop_
     _atom_identity_node
     _atom_identity_symbol     1  C  2  C
  loop_
     _attached_hydrogen_node
     _attached_hydrogen_count  1  3
save_

save_ethyl
  loop_
     _atom_identity_node
     _atom_identity_symbol     1  C  2  C  3  C
  loop_
     _attached_hydrogen_node
     _attached_hydrogen_count  1  3  2  3
save_

save_R1
  loop_
     _variable_alternative_number
     _variable_identifier_symbol
     _variable_node    1  $methyl  1   2  $ethyl  1
save_

save_carboxylic_acid
  loop_
     _atom_identity_node
     _atom_identity_symbol     1 $R1  2  C  3  O  4
O
  loop_
     _attached_hydrogen_node
     _attached_hydrogen_count  2  0  3  0  4  1
save_

loop_
     _reaction_component_number
     _reaction_component_symbol
     _reaction_component_type
            1  $carboxylic_acid  reactant
```

Fig. 5.2.2.3. Example STAR data structure where save frames encapsulate related data sets. See text for details.

The scope of data values is well defined (see Section 2.1.3.9). Only data expressed in a global block have values that are inherited in later portions of the STAR File. Data values in data blocks or save frames are restricted in scope to the current data block or save frame, respectively.

A consequence of these rules of scope and encapsulation is that a full description of the context of a STAR data value must also reflect any values carried through as global data or by de-referencing associated save frames. The results are not always intuitive.

Consider Fig. 5.2.2.3, which represents a partial description of a chemical reaction where one of the reactants is expressed as a generic structure described by the save frame `save_R1`. However, the generic structure in this case is restricted to a small number of alkyl groups, each described in its own save frame. Setting aside this prior knowledge, we see that a request for `_atom_identity_symbol` must return not only the data values in their embedded save frames, but also the save frames in their entirety and the higher-order data values that reference the matching save frames. It is only in this way that we can guarantee that the value can be used by any application. Fig. 5.2.2.4 demonstrates the full context of the returned requested data values.

Notice that the requested item occurs (among other places) in the save frame `save_carboxylic_acid` and this instance of the

```
data_reaction

save_methyl
loop_
     _atom_identity_node
     _atom_identity_symbol    1 C 2 C
loop_
     _attached_hydrogen_node
     _attached_hydrogen_count  1 3
save_

save_ethyl
loop_
     _atom_identity_node
     _atom_identity_symbol    1 C 2 C 3 C
loop_
     _attached_hydrogen_node
     _attached_hydrogen_count  1 3 2 3
save_

save_R1
loop_
     _variable_alternative_number
     _variable_identifier_symbol
     _variable_node    1  $methyl  1  2  $ethyl  1
save_

save_carboxylic_acid
loop_
     _atom_identity_symbol $R1 C O O
save_

loop_
     _reaction_component_symbol $carboxylic_acid
```

Fig. 5.2.2.4. Context for the requested values of `_atom_identity_symbol` in the preceding example. See text for details.

item is presented solely in the context of the save header and closure strings (it is shown in italics in Fig. 5.2.2.4). However, one of the values extracted from this location is the save-frame reference pointer $R1 that identifies `save_R1`, and the complete contents of this save frame are presented (because the data structure represented by the save frame *is* itself one of the values of the requested data item). Further de-referencing of the save-frame pointers within `save_R1` results in the extraction also of the complete save frames `save_methyl` and `save_ethyl`. In this example it is coincidental that there are instances of the requested data item (`_atom_identity_symbol`) within these returned save frames as well.

Notice, however, that establishing the full context of the returned data demands also that data values referencing the `save_carboxylic_acid` frame be presented. In this example, the value of `_reaction_component_symbol` at the outermost level of the data block is returned, a result that may at first seem surprising. It is only in this way that one can be sure that an arbitrary application will have access to the full semantic information carried by the data item.

Data values declared in global blocks should be presented in the same spirit of supplying the complete context in which the value was instantiated, and not simply the value in isolation. For example, given the trivial STAR File

```
global_
     _example   foo
data_1
data_2
     _example   bar
```

a request for `_example` should return the identical file, *not* the interpolated result

```
data_1
     _example   foo
data_2
     _example   bar
```

despite the latter's equivalence purely in terms of the non-contextual values returned.

5.2.3. *Star_Base*: a general-purpose data extractor for STAR Files

The stand-alone application *Star_Base* (Spadaccini & Hall, 1994) provides a facility for performing database-style queries on arbitrary STAR Files. It is generic in nature and makes no assumptions about the nature or organization of the data in a STAR File. It may indeed be used as an application-specific database tool if the user has prior knowledge of the relationships between included data items. However, by faithfully returning context as well as value in the way outlined in Section 5.2.2, it can be applied to any STAR File even without such prior knowledge.

5.2.3.1. Program features

Star_Base is a fully functional STAR File parser and may be used to test the syntactic validity of an input STAR File. It may be used to write an input STAR File directly to the output stream, while validating the structural integrity as the contents are parsed. The input format and comments are discarded on output.

Given a valid input file, *Star_Base* guarantees to write output in fully compliant STAR format.

If a data name is supplied as a request item, *Star_Base* will return the single or multiple values associated with that data name and their associated context according to the principles of Section 5.2.2, *i.e.* all loop structures, data-block headers and global headers will be returned, and save frames will be expanded as required to accommodate de-referencing of frame codes as returned values.

Where multiple data items are requested, *Star_Base* will write their occurrences to its output stream in the order they were requested, *not* in the order of appearance in the input file. This may disturb data relationships that are implicit in the ordering or association of values in the input file, but it is the responsibility of the user to track and retain such associations where they are an essential part of an application-level data model. As emphasized before, a generic STAR tool will make no assumptions about data models and will simply return values and contexts as requested.

To illustrate the effect of this, consider a request for the following data items from the example file of Fig. 5.2.2.1:

```
_basis_set_atomic_name
_basis_set_atomic_symbol
_basis_set_contraction_scheme
```

Star_Base will return the following result:

```
data_Gaussian
loop_
     _basis_set_atomic_name
     _basis_set_atomic_symbol
    loop_
     _basis_set_contraction_scheme
    stop_

    hydrogen H
        (2)->[2]  (2)->[2]  (2)->[1]  (3)->[2]  stop_

    lithium Li
        (4)->[4]  (9,4)->[3,2]  (4,3)->[3,2]  stop_
```

However, if the same items are requested in a different order,

```
_basis_set_atomic_name
_basis_set_contraction_scheme
_basis_set_atomic_symbol
```

the result is structured differently:

```
data_Gaussian
loop_
     _basis_set_atomic_name
    loop_
     _basis_set_contraction_scheme
    stop_
     _basis_set_atomic_symbol

    hydrogen
        (2)->[2]  (2)->[2]  (2)->[1]  (3)->[2]  stop_
    H
    lithium
        (4)->[4]  (9,4)->[3,2]  (4,3)->[3,2]  stop_
    Li
```

In the examples so far, one or more data items have been requested by name. *Star_Base* extends the type of requests that can be made through its own query language. This gives it much of the power of a database query language such as SQL. Three types of query are supported, known as *data*, *conditional* and *branching* requests.

A *data request* is a straightforward generalization of the request by data name. Individual data items may be requested by name, as may individual data blocks or save frames. Wild carding is permitted to generalize the requests. More details are given in Section 5.2.3.2.

A *conditional request* involves one or more conditions; only data items satisfying the conditions are returned. More details are given in Section 5.2.3.3.

A *branching request* applies similar conditions to establish the context in which matching data items occur within the file, but may also apply scoping rules to select among the available contexts. Only data items matching both the conditions imposed on their values *and* the requested scope are returned. It is the existence of such branching conditions that gives *Star_Base* the ability to select data matching the specific requirements of overlying data models. Again, however, it is emphasized that the program itself operates without any semantic awareness of the significance of the data that is implied within the overlaid data model. More details on branching requests are given in Section 5.2.3.4.

5.2.3.2. The *Star_Base* data request

A *data request* is the simplest type of query used to extract single items from a file. It may be formed from any of the following string types:

(i) a **name** string, *e.g.* `_atom_identity_symbol`;

(ii) a **block** string, *e.g.* `data_Gaussian`;

(iii) a **frame** string, *e.g.* `save_methyl`.

In accordance with the principles set out earlier in this chapter, data requests satisfy the following rules:

(i) Requested data items are returned with their associated context (*i.e.* including the headers of any containing data blocks, save frames and loop structures).

(ii) A request for a data block returns all preceding global blocks (since the data block will contain by inheritance all values in the global blocks).

(iii) A request for a save frame also returns the header of the data block encompassing the save frame. All frame-pointer codes are resolved so that if a requested save frame contains pointer codes to other save frames, these are also returned.

(iv) A request for `global_` returns all global blocks, together with all data-block headers in their scope.

(v) The request need not be specified explicitly. Two wild-card characters are permitted. An asterisk (*) represents *any sequence*

Table 5.2.3.1. *Permitted constructions for a Star_Base conditional request*

<data request>
<data request> <operator> <text string>
<conditional request> & <conditional request>
<conditional request> | <conditional request>
!<conditional request>

of characters and a question mark (?) represents *any single* character.

(vi) A request for looped data returns the items in the order requested, making any necessary adjustments to the structuring of nested loops to preserve the original context.

(vii) A request for data within a save frame returns those items plus the associated context.

(viii) If a requested data item includes a save-frame pointer as a value, the referenced save frame is returned intact. All other pointers contained within the returned data are resolved.

(ix) A request for a data item in a global data block will also return the data-block headers within the scope of the global block.

(x) The scope of a data request is *the entire input file*. Control of the search scope is only possible within branching requests.

5.2.3.3. The *Star_Base* conditional request

While a data request allows retrieval of data items according to *name*, conditional requests allow retrieval of data items by *value*. The general form of a conditional request may be characterized as <data request><operator><text string>, where <data request> is any data request as defined in the preceding section, <operator> is any of the test operators defined below, and <text string> is a string pattern against which values of data items retrieved by the data request are matched according to the operator specified.

Conditional requests may be combined by set operators &, | and ! to provide logical AND, OR and NOT tests. Table 5.2.3.1 lists the allowed constructions for a conditional request. A bare data request is considered a degenerate case of a conditional request.

The construction <conditional request> & <conditional request> allows for the *conjunction* of conditionals. All data are returned (including context) from the intersection of sets of data that individually satisfy the conditions to be a non-empty set.

It is important to note that the conjunction of conditionals based on different data names is the empty set.

The construction <conditional request> | <conditional request> allows for the *disjunction* of conditionals. All data are returned (including context) from the union of sets of data that individually satisfy the conditions to be a non-empty set.

The construction !<conditional request> allows for the *negation* or *complement* of conditionals. All data are returned (including context) from the universal set of data that do not satisfy the conditions of the conditional request. The universal set is defined as the input file.

Table 5.2.3.2 lists the permitted value-matching operators when a retrieved data value is compared with a target text string in the basic test <data request><operator><text string> described above. (If the <text string> contains white-space characters, it must be quoted with matching single or double quotes. The test is performed on the value of the text string, *i.e.* the complete text string including white-space characters but omitting the surrounding quote characters.)

Two classes of operators are defined. *Text operators* may be used to test for string equality, substring containment or greater and lesser values (where the 'greater' and 'lesser' values for text strings are based on the ASCII character set ordering sequence).

Table 5.2.3.2. *Value-matching operators in Star_Base conditional requests*

Requests are of the form <data request> <operator> <text string>. The second column describes the relationship that data identified by the <data request> must satisfy against the <text string> in order to be returned as part of the result set.

Operator	Relationship
Text comparison operators:	
∼=	Is identically equal to
?=	Includes as a substring
∼<	Is less than (in ASCII order)
∼>	Is greater than (in ASCII order)
∼!=	Is not identically equal to
?!=	Does not include as a substring
∼<=	Is not greater than (in ASCII order)
∼>=	Is not less than (in ASCII order)
Numerical comparison operators:	
=	Is equal to
<	Is less than
>	Is greater than
!=	Is not equal to
<=	Is not greater than
>=	Is not less than

These tests are valid for any STAR application. *Numerical operators* permit comparison of the numerical values implied by the returned data-value strings. Recall from Chapter 2.1 that data values in STAR are specified only as character strings. Casting to different types may be performed by specific applications, but is *not* defined for arbitrary STAR applications. Nevertheless, *Star_Base* recognizes that a majority of STAR applications will in fact specify numeric types, and therefore allows for numerical comparisons based on interpretations of certain value strings according to the conventions adopted by CIF for the **numb** data type (Section 2.2.7.4.7.1). Such values may be given as integers, real numbers or in scientific notation.

5.2.3.4. The *Star_Base* branching request

Both conditional and data requests will retrieve matching data items wherever they may be found in the input file; the scope of the query in both cases is the entire file.

The top-level query type supported by *Star_Base*, the *branching request*, allows selection of sub-requests based on the results of prior tests, and also allows the narrowing or expansion of the scope of a request. The effect is to permit extensive control over the selection of data matching complex conditions. It is this which gives *Star_Base* the power of a database query language.

Note again that the user will in general need prior knowledge of the arrangement of data items within a STAR File in order to compose meaningful requests; *Star_Base* is agnostic about the organization and structure of the contents of a data file and will simply return exactly those data items and their context that match the specified conditions.

A branching request takes the following form:

if_ <condition> <branch_request>
  [**else_** <branch_request>]
  [**unknown_** <branch_request>]
endif_

The <condition> has exactly the same form as a conditional request, but does not return data to the calling process. It returns only a logical value that is used to determine which branch to evaluate. This logical return value may be TRUE if the condition is satisfied, UNKNOWN if the condition is not satisfied because there was no occurrence of a requested data name within the current

scope of the query, or FALSE if the condition is not met otherwise.

The branching request must have a condition and a branch request that is made if the condition returns TRUE. The other possible branches are optional. The **unknown_** branch, if present, is executed when the condition returns a value of UNKNOWN. If there is no **unknown_** branch, then the default truth value is set as FALSE (*i.e.* a return value of UNKNOWN is treated as equivalent to FALSE) and the **else_** branch is executed if present. It is possible to override this behaviour by using the special operator **assume_true_** before a condition. This operator forces the default truth value to TRUE when a condition returns an UNKNOWN value. It is a useful shorthand when the same branch request is applied against a condition that is either TRUE or UNKNOWN. The syntax is

assume_true_ (<condition>).

A <branch request> (that is, the set of actual individual requests within a branching request construct) has three possible forms:

(i) a conditional request,

(ii) a branching request,

(iii) **scope_**<scope setting> <branch request> **endscope_**.

Note carefully the different contexts in which branch requests and nested branching requests may occur.

scope_<scope setting> specifies the range of data to be searched in the input file. The effect of the setting is closed by the **endscope_** statement. The permitted values of <scope setting> are:

(i) **data_item_** restricts the branch request to the data items in the condition,

(ii) **loop_packet_** restricts the branch request to the contents of the loop packet in which data match the condition,

(iii) **loop_structure_** restricts the branch request to the loop structure in which data match the condition,

(iv) **save_frame_** restricts the branch request to the contents of the save frame in which data match the condition,

(v) **data_block_** restricts the branch request to the contents of the data block in which data match the condition,

(vi) **file_** specifies that the branch request applies to the contents of the file containing data matching the condition (the default setting).

The default scope is invoked when a **scope_**<scope setting> is *not* specified; in such a case the scope of the branch request is the same as that of the condition.

Fig. 5.2.3.1 demonstrates the construction of a branching request that restricts the scope of the query. The two requests in this figure are applied to the STAR File example of Fig. 5.2.2.1. In Figure 5.2.3.1(*a*), the query is targeted to retrieve all data items in a loop packet where the value of `_basis_set_contraction_scheme` includes the substring `(3)`, provided that the value of `_basis_set_atomic_name` identically matches the string value `hydrogen` in the next outer nested loop packet. The data relevant to the contraction scheme labelled `(3)->[2]` are returned. Note how the wildcard data request `_*` retrieves the data items from the next outer loop structure in which the requested data lie.

In the example of Fig. 5.2.3.1(*c*), the request for an unknown data name cannot be matched within the input file, and the **unknown_** branch of the request is executed. In this case, the secondary request is more specific (only data names including the substring `contraction` are matched) and hence only a few items from the second-level loop are returned. Scopes can be expanded or contracted. If the example of Fig. 5.2.3.1(*a*) were to be modified by replacing the innermost `scope_loop_packet_` declaration with `scope_loop_structure_`, the query would proceed by testing for the existence of a contraction scheme value including the string `(3)` in the loop packet relevant to the hydrogen results (as before). Finding that this condition was satisfied, the result returned would be all data names in

```
if_ _basis_set_atomic_name  = hydrogen
  scope_loop_packet_
  if_ _basis_set_contraction_scheme ?= (3)
    scope_loop_packet_
     _*
    endscope_
  endif_
  endscope_
endif_
```

(*a*)

```
data_Gaussian
loop_
     _basis_set_atomic_name
     _basis_set_atomic_symbol
     _basis_set_atomic_number
     _basis_set_atomic_mass
    loop_
     _basis_set_contraction_scheme
     _basis_set_funct_per_contraction
     _basis_set_primary_reference
     _basis_set_source_exponent
     _basis_set_source_coefficient
     _basis_set_comments_index
     _basis_set_atomic_energy
        loop_
     _basis_set_function_exponent
     _basis_set_function_coefficient
        stop_
    stop_

    hydrogen H 1 1.0079
        (3)->[2] 2:1 PKC1.23.1 R75 R75 C13,C19 -0.496979
            4.5018000E+00 1.5628500E-01
            6.8144400E-01 9.0469100E-01
            1.5139800E-01 1.0000000E+01
            stop_
        stop_
```

(*b*)

```
if_ _basis_set_atomic_name  = hydrogen
  scope_loop_packet_
  if_ _basis_set_contraction_xxxxxx ?= (3)
    scope_loop_packet_
     _*
    endscope_
  unknown_
     _*contraction*
  endif_
  endscope_
endif_
```

(*c*)

```
data_Gaussian
loop_
    loop_
     _basis_set_contraction_scheme
     _basis_set_funct_per_contraction
    stop_

        (2)->[2] 1:   (2)->[2] 1:
        (2)->[1] 2    (3)->[2] 2:1
        stop_
```

(*d*)

Fig. 5.2.3.1. Examples of branching requests and the results returned by *Star_Base* from the example file of Fig. 5.2.2.1. (*a*) A query designed to extract a data structure relevant to one contraction scheme and one atom type. (*b*) The results of that request. (*c*) A similar request, but with a branch followed when the condition cannot be matched against a requested data item in the current scope, and (*d*) the resulting output. See text for discussion.

the encompassing loop *structure – i.e.* in this particular example, the complete loop contents would be returned.

Note that *Star_Base* faithfully returns context even in the processing of complex branching requests. Therefore if, for example, a save-frame pointer is returned as a data value following the processing of a request, the associated save-frame contents will be returned in full so that they are referenced in the returned STAR data structure.

5.2.3.5. Implementation issues

Star_Base is implemented in the C programming language, and exploits Gnu's *flex* and *bison* compiler-compiler system to generate a lexer and parser for the STAR File and a separate lexer and parser for the *Star_Base* query language.

The STAR File parser builds an *in-memory* representation (much like most programming-language compilers) of the file contents, and differs from similar applications that are based on a single pass over a stream (like SAX for XML applications).

While a system like *CIFtbx* retains a block copy of the STAR File in memory, the initial *Star_Base* processing removes all comments and formatting, and stores the meaningful tokens in a binary tree representation. For each STAR File container (global block, data block, save frame, loop or data item) there is a C structure defined. For each of these there are additional structures defined that hold sequences of containers. The nodes of this tree are populated with these structures. Each leaf of the tree is the data item consisting of the data name and its associated value. A binary tree of the global-block sequences is built in reverse order (that is, in an order reverse to that in which they appear in the file), making it simple to identify the global values in scope for a specific data block. It will be recalled that the STAR File semantics require a backward scan through the file to pick up the global blocks in scope.

The binary search algorithm employed is the classic *tsearch* of Knuth (1973), which is part of the standard C libraries. Given modern computer systems, the implementation is extremely fast and efficient. There are no files in existence whose size would test the limits of *Star_Base*.

The use of a binary tree simplifies the process by which a legitimate STAR File is returned as output by *Star_Base* and also how the scope over which the conditionals operate can be controlled by the user. The program stores references to the data nodes of the tree it needs to extract when outputting. Since the location in the original data tree is always stored, the program is easily able to reconstruct the correct structure of the file by walking the tree, identifying the nodes that need to be output in addition to the data.

Star_Base is by default the 'gold standard' for testing other applications for correctness with respect to the syntax and semantics of the STAR File. It can be said that the output of *Star_Base* is not optimal, since it is yet another STAR File and one which is devoid of the original comments and formatting. However *Star_Base* is in essence an API for STAR File applications, rather than a stand-alone program (although it is often used in that way). *Star_Base* was the platform from which BioMagResBank's *starlib* (Section 5.2.6.4) was developed.

5.2.4. Editing STAR Files with *Star.vim*

The *vim* editor supports syntax highlighting for a wide variety of languages through syntax definition rules. The definition rules for STAR Files are simple and small in number. The entire syntax is defined by 19 rules that include the regular expressions for the STAR File keywords, data names, numbers, single- and double-quoted strings, semicolon-delimited text and frame codes. The language for defining a syntax in *vim* is very simple and very powerful. The constructs allow the user to define the syntax precisely enough so that the system does not match patterns within other patterns, unless directed to.

There are three types of syntax items: **keyword**, **match** and **region**.

keyword can only contain keyword characters, no other syntax items. It will only match with a complete word (there are no keyword characters before or after the match). The rule for the STAR syntax is

```
syn keyword strKeyword   global_ save_ loop_ stop_
```

match is a match with a single regular expression (regexp) pattern. The rule for matching on a save-frame code is

```
syn match strFramecode  "$[^ \t]\+"
```

region starts at a match of the 'start' regexp pattern and ends with a match with the 'end' regexp pattern. Any other text can appear in between. There can be several 'start' and 'end' patterns in the one definition. A 'skip' regexp pattern can be used to avoid matching the 'end' pattern. There are a number of character offset parameters that allow the user to redefine the start and end of the matched text given the pattern that matches the regular expression. Quite separately, one can define the region for highlighting, which can be different from the matched text.

The rule that matches a double-quoted string is

```
syn region strString start=+^"+ start=+\s\+"+ms=e
          end=+" +me=e-1 end=+"\t+me=e-1 end=+"$+
           contains=strSpecial " skip=+\\\\\|\\"+
```

In this rule, the beginning of the pattern is a double quote that is either the first character on the line or that has one or more white-space characters before it. The beginning of the matched string (*ms*) is the end of the matched pattern (*e*). That is, the matched string begins at the quote. The end pattern is a double quote followed by a single space, a tab or an end of record. The end of the matched string (*me*) is one character less than the end of the matched pattern (*e*). That is, the trailing character after the closing double quote is not considered part of the matched string. By default the characters between location *ms* and *me* are highlighted. This too can be controlled, and by including $hs = ms + 1$ and $he = me - 1$ the highlighted text would not include the delimiting double quotes.

As these rules are based on regular expressions, there is no possibility of using them to validate the STAR File structure. However, problems in the structure are often identifiable by unexpected or irregular highlighted text [a fact often used in graphical CIF editors to help the user locate visually errors in syntax (see *e.g.* Section 5.3.3.1.4)].

5.2.5. Browser-based viewing with *StarMarkUp*

StarMarkUp is a Tcl/Tk program that takes any STAR File as input and outputs the contents as HTML. The output is a faithful copy of the input, and there is no reformatting or deletion of content.

During transformation, the contents can be cross-referenced against any other STAR File using HTML anchors. This feature is particularly useful when marking up a data file, since the data names contained within can be hyperlinked to their definition in their dictionary. Furthermore, the definitions contained within the dictionary can be hyperlinked to the DDL dictionary. *StarMarkUp* makes no presumptions about the version of DDL employed, the preferred dictionary structure or the specific application

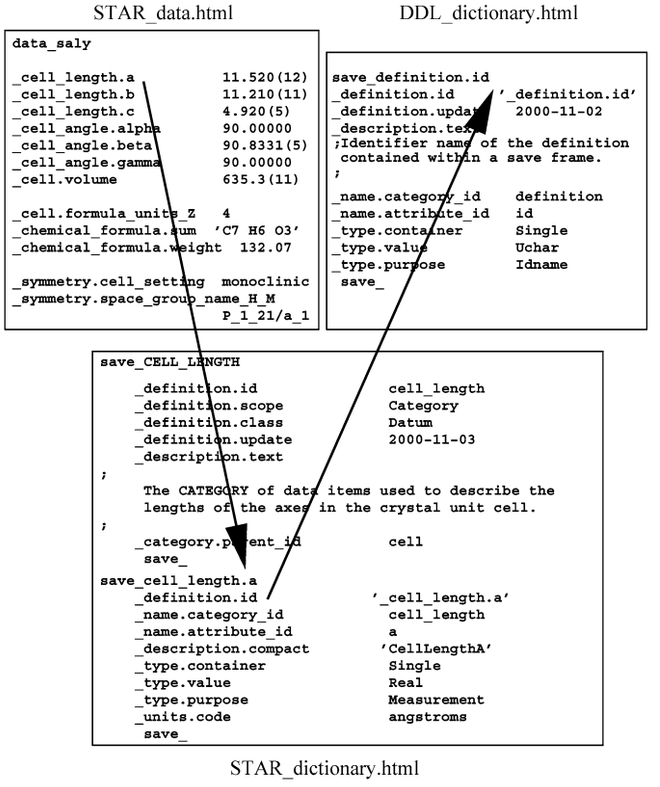

Fig. 5.2.5.1. Hyperlinking markup generated by *StarMarkUp*. The example is based on an extended relational dictionary definition language, StarDDL.

of the STAR File. The program understands only the rules of a legitimate STAR File.

StarMarkUp provides a number of hyperlinking facilities. During markup, it automatically hyperlinks frame-code values to the internal save-frame block to which they point. As part of the same process, *StarMarkUp* inserts anchors in all data and save-frame blocks. It does this in anticipation that there may be the need to hyperlink to these anchors from another STAR File. The most obvious application for this is in the marking up of the DDL dictionary. The list of anchors generated from this process can be passed on to the discipline dictionary during its markup phase (Fig. 5.2.5.1). In this way, the tags used in the discipline dictionary to define the data names can be made to point back to their entry in the DDL dictionary. At the same time that the discipline dictionary is being marked up, a list of its anchors is being generated (each anchor being to a data-item definition). This list is used when marking up an instance of the discipline STAR data file to hyperlink each data name back to its definition in the discipline dictionary.

StarMarkUp comprises a hand-crafted tokenizer employing a character buffer and one-character look-ahead to identify accepted tokens. *StarMarkUp* does not build an internal representation of the file, but functions as a streaming parser. The *GetToken()* function returns a structure consisting of the token type (an enumerated set) and the token value (the lexeme associated with that token). In most cases, the lexeme is marked up and injected into the output stream. If the token is associated with a STAR File block, the parser will recursively call a *MarkUpBlock()* function. A recursive descent parser makes it very easy to treat all STAR File blocks (**`global_`**, **`data_`** and **`save_`**) in identical fashion.

StarMarkUp is implemented in Tcl/Tk for novelty and not because of any particular superior qualities of the language and its API. It is, however, fast and sufficiently flexible, and extensions to the program can be rapidly implemented and tested.

5.2.6. Object-oriented STAR programming

The STAR File syntax is very simple but flexible, and suggests a number of well defined data structures. 'Scalar' values (*i.e.* single text strings identified by specific data names) are obvious, as are 'vectors' (multiple string values associated with a data name in a **`loop_`** declaration). 'Matrices' are constructed by associating identical-length vector items in the same **`loop_`** structure. Note that these data structures are not true matrices, but arrays or tables of column-addressable vectors. Column addressing arises from the need to associate each set of values with a distinct data name in the loop header declaration.

Through the nested loop construct in STAR, vector or matrix *elements* are permitted within vectors or matrices. Save frames provide addressable data structures containing one or more scalar, vector or matrix components; and data blocks contain similar data structures with the inclusion of embedded save frames.

This hierarchy of structure allows many other data models to be mapped onto the STAR syntax. For example, STAR does not in itself provide addressable *rows* in matrix structures, but such a concept may be achieved in application-specific ways. For example, the relational database structure of mmCIF and other DDL2 applications ensures that table rows can be identified by specific key values. Loops of multiple values associated with a single data name are similar to *lists* or one-dimensional *arrays* as defined in many computer languages. Simple *associative arrays*, or 'hashes', such as are used to great effect in Perl and Python, can be modelled by designating the first value of a two-item loop structure as the 'key' item, by analogy with the relational database example above (in such an application the key value of course needs to be unique). The hierarchy of nested loops can be used to model certain types of complex structures such as might be defined in a C-language 'struct', for example.

There is a natural possibility of representing particular well defined data structures as 'objects' and handling STAR Files by object-oriented programming techniques. Where applications associate properties of specific data items with external reference descriptions, as in the DDL dictionaries, the dictionary entries can be considered to express types, relationships and even methods associated with the object classes.

However, the versatility of the STAR syntax means that there are many ways of designing STAR objects and their relationships, and it is likely that the most useful efforts will be in designing applications for specific purposes or subject areas. Below we describe a number of prototypes and implementations of object-oriented STAR programming. Some are rather generic in outlook; others are informed by the crystallographic viewpoint embodied in CIF.

5.2.6.1. *OOSTAR*

The *OOSTAR* approach (Chang & Bourne, 1998) developed and tested an Objective-C (Pinson & Richard, 1991) toolset for STAR representation. The developers selected Objective-C over the more common C++ language because of its perceived advantages of (i) loose data typing, (ii) run-time type checking and (iii) message-passing ability. These were all seen to aid flexible software design, an asset for applications built on the very flexible structure of the STAR syntax itself.

Chang & Bourne constructed a set of classes built on the basic objects *item* and *value*. Their model builds upwards through a hierarchy of classes describing relationships between objects. The *ItemAssoc* class contains data members corresponding to the data item itself, its included value or set of values, and the data name that is used to reference the item; the class also has pointers to

previous and next elements to allow iteration. The *StarAssoc* class contains methods for manipulating STAR objects through setting and retrieval of data names and assignment of values. *DataBlock* is the class containing all the lower-order items and associations, and one or more such *DataBlock* classes comprise the *StarFile* class.

The *OOSTAR* approach also provides *Dictionary* and *DictionaryElement* classes so that STAR applications that do describe element attributes in external dictionary files can make use of such information.

Some sample applications were built with the class libraries of the *OOSTAR* toolset, and they are described in Chang & Bourne (1998). They include: simple converters of STAR data files to HTML pages with hyperlinks to associated dictionary entries; a query tool for retrieving the data values for a specific item specified by name within a specific data block; and a query mechanism that retrieves a set of items from a STAR loop structure and represents their values in an array, *i.e.* flattening if necessary any nested loops into a purely tabular presentation. These query tools are different from *Star_Base* primarily in that they do not return the context in which the data are found in the original file – their interpretation depends on a detailed *a priori* understanding of the target file structure.

This approach is rather different from the canonical description of STAR given in this volume, and was never developed into full-blown applications (in part because the developers recognized that the Java language would provide a preferable platform for further development). Nevertheless, it provides interesting ideas for the developer considering building object-oriented applications for STAR Files.

5.2.6.2. *CIF++*

CIF++ was a small library of classes designed by Peter Murray-Rust during the time that DDL was being developed as a language for representing the properties and attributes of STAR data items (*e.g.* Murray-Rust, 1993). The purpose of this project was to demonstrate the design of data classes that represented real-world objects such as molecules and crystal cells. At a time when the structure of CIF was under intense discussion, many of the classes were potentially extensible to STAR features that were a superset of those found in CIF.

While these classes were never developed into fully functional applications, they demonstrated a potentially fruitful approach to model representation and gave rise to the idea of including methods in STAR dictionary definitions, thus allowing STAR applications to dynamically associate algorithmic relationships and operations with data objects by parsing the methods description in the dictionary definitions. This approach is currently being developed into a relational expression language for STAR called dREL (Spadaccini *et al.*, 2000).

The ideas behind *CIF++* were further developed in a class library for molecular representation called *Democritos*, and have informed the representation of molecular and crystal structure in Chemical Markup Language (CML) and in the structured document browser *Jumbo* (Murray-Rust, 1998).

The *CIF++* classes have been refactored into Java using the W3C document object model (W3C, 2004) and other DOM-like models. They are now based on an XML schema which is the abstraction of the formal DDL1 specification (Chapter 2.5).

These are available as part of the *Jumbo* distribution at http://cml.sf.net. The design involves an interface that would allow the C++ classes to be recreated from the Java.

5.2.6.3. *CIFOBJ*

The *CIFOBJ* class library (Schirripa & Westbrook, 1996) was developed to provide an object view of the mmCIF dictionary and to complement the relational *CIFLIB* class library (Westbrook *et al.*, 1997) for handling mmCIF data. As such, it is very much tuned to crystallographic applications and it handles only the subset of STAR features used in CIF dictionaries. This does, however, include some limited handling of save frames, and is therefore a little more complex than applications interested only in CIF data input/output processing.

CIFOBJ has two components. The first builds a persistent store of objects of types item, subcategory, category and dictionary. Each such object is a container for all relevant attributes permitted for that object type. The object store is populated from the mmCIF dictionary by the *CIFOBJ* loader class (using methods provided by *CIFLIB*). This loader class assembles the dictionary objects and passes them to an object-storage manager. The second component of the *CIFOBJ* class library provides the methods necessary for building dictionary objects from the persistent store and for passing attribute strings to procedures concerned with establishing the integrity of data values.

The main reason for discussing this implementation here is to indicate the rapid growth in complexity needed to impose an application-specific object view on even a relatively simple STAR data structure. In generic prototype STAR projects such as *OOSTAR*, half a dozen or so classes and associated methods suffice to represent the highest-level abstract concepts implied in STAR constructions. In *CIFOBJ*, however, dozens of methods are associated with dictionary access. Dictionary-driven validation of an mmCIF numerical data value can involve:

(i) retrieval from the dictionary of the extended type declaration associated with the data item;

(ii) validation of the basic type (*i.e.* that it is indeed numeric) against the primitive data types supported by the dictionary;

(iii) validation of the extended type by regular-expression matching of the string representation of the value against the allowed patterns stored in the dictionary;

(iv) retrieval of any existing range constraints specified in the dictionary and comparison with the data value;

(v) location and evaluation of any associated standard uncertainty;

(vi) identification of the units in which the physical quantity is expressed;

(vii) if necessary, conversion of the units according to the conversion tables stored in the dictionary.

More complexity arises from the relationships between data items expressed through dictionary attributes such as name aliasing, parent–child dependencies and category membership.

Nevertheless, the complexity of these relationships is an indication of the richness of the metadata available through the dictionary approach, and the availability of well defined object representations simplifies the construction of well designed large-scale application frameworks, such as underpin the Protein Data Bank (Berman, Battistuz *et al.*, 2002) and Nucleic Acid Database (Berman, Westbrook *et al.*, 2002).

5.2.6.4. *starlib*

BioMagResBank (BMRB) is a repository for NMR spectroscopy data on proteins, peptides and nucleic acids at the University of Wisconsin – Madison (Ulrich *et al.*, 1989). For some time, NMR data sets have been exchanged within this environment using STAR Files; an NMRStar data dictionary to define the data names used for tagging NMR data is under development

(BioMagResBank, 2004). The *starlib* class library was developed at BMRB for handling NMRStar files, but its initial application to such files independently of the prototype data dictionary means that it is applicable to any STAR File. It does not provide a relational database paradigm (although this is a long-term goal). However, it does provide objects and methods suitable for searching and manipulating STAR data.

Table 5.2.6.1 lists the top-level classes used in *starlib*. *ASTnode* is a formal base class, providing the types and methods that can be used in other derived classes. *StarFileNode* is the root parent of all other objects contained in an in-memory representation of a STAR File; in practice it contains a single *StarListNode*, which is the list of all items contained in the file. *BlockNode* is a class which contains a partition of the STAR File: the class handles both data blocks and global blocks. Data-block names are stored in instances of the *HeadingNode* object, which also holds save-frame identification codes and is therefore useful for accessing named portions of the file.

DataNode is a virtual class representing the types of data objects handled by the library (accessed directly as *DataItemNode*, *DataLoopNode* and *SaveFrameNode*).

Looped data items are handled by a number of objects. *DataLoopNameListNode* is a list of lists of names in a loop. The first list of names is the list of names for the outermost loop, the second list of names is the list of names for the next nesting level and so on. *LoopNameListNode* is a list of tag names representing one single nesting level of a loop's definition. *LoopTableNode* is a table of rows in a *DataLoopNode* (not itemized in Table 5.2.6.1; it is an object representing a list of tag names and their associated values, a particular case of *DataNode*). *starlib* views a loop in a STAR file as a table of values, with each iteration of the loop being a row of the table. Each row of the table can have another table under it (another nesting level), but such tables are the same structure as the outermost one. Thus *LoopTableNode* stores a table at some arbitrary nesting level in the loop. A simple singly nested loop will have only one loop table node, but a multiply nested loop will have a whole tree of loop tables. *LoopRowNode* is a single row of values in a loop.

DataNameNode holds the name of a tag/value pair or a loop tag name. *DataValueNode* is the type that holds a single string value from the STAR file and the delimiter type that is used to quote it.

DataListNode and *SaveFrameListNode* store lists of data within higher-order data objects or save frames, and are internal classes rarely invoked directly by a programmer.

A number of observations may be made regarding this approach. Firstly, the objects can be mapped with reasonable fidelity to the high-level Backus–Naur form representation of STAR (Chapter 2.1). Secondly, it is computationally convenient to abstract common features into parent classes, so that, for example, individual data items, looped data and save frames are represented as child objects of the *DataNode* object, and not themselves as first-generation children of the base class. Thirdly, the handling of nested loops may be achieved in different ways; *starlib* has chosen a particular view that is perhaps well suited to relational data models.

As expected within a programming toolkit, *starlib* offers a large number of methods for retrieving STAR data values, adding new data items, extending or re-ordering list structures, and performing structural transformations of the in-memory data representation. Unlike the stand-alone *Star_Base* application, it does not guarantee that output data will be in a STAR-conformant format; and the programmer is left with the responsibility of validating transformed data at a low level.

Table 5.2.6.1. *Object classes for manipulating STAR data in starlib*

ASTnode	The base class from which all other classes are derived
StarFileNode	The STAR File object
StarListNode	List of items contained in the STAR File
BlockNode	A data or global block
HeadingNode	Labels for major STAR File components
DataNode	General class for data objects
DataLoopNameListNode	List of lists of names in a loop
LoopNameListNode	List of tag names representing one nesting loop level
LoopTableNode	Table of rows in a loop
LoopRowNode	Single row of values in a loop
DataNameNode	A data name
DataValueNode	A single string value
DataListNode	List of data within a higher-order data object
SaveFrameListNode	List of data items allowed in a save frame

Nevertheless, this is a substantial and important library which, as with *CIFOBJ*, has played an important role in the functioning of a major public data repository. Development of the class libraries continues, with a Java version now available.

5.2.6.5. *StarDOM*

A convenience of well designed object representations is that effective transformation between different data representations may be possible. The *StarDOM* package (Linge *et al.*, 1999) demonstrates a transformation from STAR Files to an XML representation, where the tree structure of a STAR File as interpreted in the *starlib* view above is mapped to a document object model (DOM; W3C, 2004). This approach is similar to *Jumbo*, mentioned above in Section 5.2.6.2.

A demonstration of *StarDOM* is the transformation of the complete set of NMR data files at BioMagResBank to XML. The resultant files can then be interrogated using the *XQL* query language (Robie *et al.*, 1998). In this example implementation, the target XML document type definition (DTD) includes a small number of XML elements matching the STAR objects *global* and *data block*, *save frame*, *list*, *data item*, *data name* and *data value*. Particular data names are recorded as values of the <NAME> element. The authors of the *StarDOM* package are considering an extension in which named data items map directly to separate XML elements; the goal is to develop an NMR-specific DTD that is isomorphous to the emerging NMRStar data dictionary.

References

Berman, H. M., Battistuz, T., Bhat, T. N., Bluhm, W. F., Bourne, P. E., Burkhardt, K., Feng, Z., Gilliland, G. L., Iype, L., Jain, S., Fagan, P., Marvin, J., Padilla, D., Ravichandran, V., Schneider, B., Thanki, N., Weissig, H., Westbrook, J. D. & Zardecki, C. (2002). *The Protein Data Bank. Acta Cryst.* D**58**, 899–907.

Berman, H. M., Westbrook, J., Feng, Z., Iype, L., Schneider, B. & Zardecki, C. (2002). *The Nucleic Acid Database. Acta Cryst.* D**58**, 889–898.

BioMagResBank (2004). *NMR-STAR Dictionary*. Version 3.0. (Under development.) http://www.bmrb.wisc.edu/dictionary/3.0html/.

Chang, W. & Bourne, P. E. (1998). *CIF applications. IX. A new approach for representing and manipulating STAR files. J. Appl. Cryst.* **31**, 505–509.

Knuth, D. E. (1973). *The art of computer programming.* Vol. 3, *Sorting and searching*, pp. 422–447. Reading, MA: Addison-Wesley.

Linge, J. P., Nilges, M. & Ehrlich, L. (1999). *StarDOM: from STAR format to XML. J. Biomol. NMR*, **15**, 169–172.

Murray-Rust, P. (1993). *Analysis of the DDL/dictionary parsing problem. Proc. First Macromolecular Crystallographic Information File (CIF) Tools Workshop*, ch. 12. New York: Columbia University.

Murray-Rust, P. (1998). *The globalization of crystallographic knowledge*. *Acta Cryst.* D**54**, 1065–1070.

Pinson, L. J. & Richard, R. S. (1991). *Objective-C object-oriented programming techniques*. Reading, MA: Addison-Wesley.

Robie, J., Lapp, J. & Schach, D. (1998). *XML Query Language (XQL)*. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

Schirripa, S. & Westbrook, J. D. (1996). *CIFOBJ. A class library of mmCIF access tools.* Version 3.01 reference guide. Technical Report NDB-269. Rutgers University, New Brunswick, New Jersey, USA.

Spadaccini, N. & Hall, S. R. (1994). *Star_Base: accessing STAR File data. J. Chem. Inf. Comput. Sci.* **34**, 509–516.

Spadaccini, N., Hall, S. R. & Castleden, I. R. (2000). *Relational expressions in STAR File dictionaries. J. Chem. Inf. Comput. Sci.* **40**, 1289–1301.

Ulrich, E. L., Markley, J. L. & Kyogoku, Y. (1989). *Creation of a nuclear magnetic resonance data repository and literature database. Protein Seq. Data Anal.* **2**, 23–37.

W3C (2004). *Document Object Model (DOM).* http://www.w3.org/DOM/.

Westbrook, J. D., Hsieh, S.-H. & Fitzgerald, P. M. D. (1997). *CIF applications. VI. CIFLIB: an application program interface to CIF dictionaries and data files. J. Appl. Cryst.* **30**, 79–83.