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