International Tables for Crystallography (2006). Vol. G, Section 5.2.5, pp. 494–495.

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=+\\\\\|\\"+
```

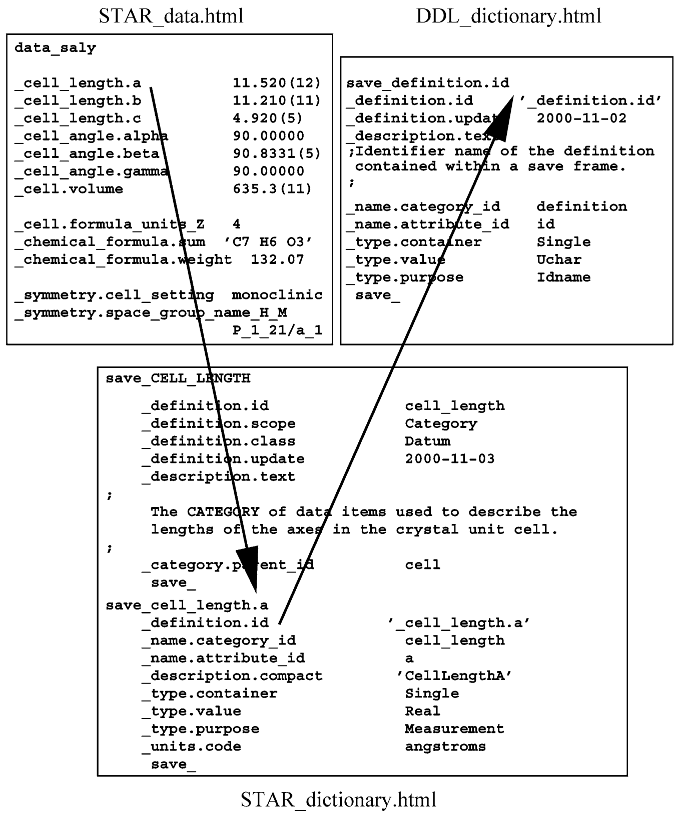
In this rule, the beginning of the pattern is a double quote that is either the first character on the line or that has one or more white-space characters before it. The beginning of the matched string (*ms*) is the end of the matched pattern (*e*). That is, the matched string begins at the quote. The 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



```
data_saly

_cell_length.a            11.520(12)
_cell_length.b            11.210(11)
_cell_length.c            4.920(5)
_cell_angle.alpha         90.00000
_cell_angle.beta          90.8331(5)
_cell_angle.gamma         90.00000
_cell.volume              635.3(11)

_cell.formula_units_Z    4
_chemical_formula.sum   'C7 H6 O3'
_chemical_formula.weight  132.07

_symmetry.cell_setting  monoclinic
_symmetry.space_group_name_H_M
                          P_1_21/a_1
```

```
save_definition.id
_definition.id         '_definition.id'
_definition.update      2000-11-02
_description.text
;Identifier name of the definition
 contained within a save frame.
;
_name.category_id      definition
_name.attribute_id     id
_type.container        Single
_type.value            Uchar
_type.purpose          Idname
 save_
```

```
save_CELL_LENGTH

    _definition.id                  cell_length
    _definition.scope               Category
    _definition.class               Datum
    _definition.update              2000-11-03
    _description.text
;
     The CATEGORY of data items used to describe the
     lengths of the axes in the crystal unit cell.
;
    _category.parent_id             cell
     save_
save_cell_length.a
    _definition.id                 '_cell_length.a'
    _name.category_id               cell_length
    _name.attribute_id              a
    _description.compact           'CellLengthA'
    _type.container                 Single
    _type.value                     Real
    _type.purpose                   Measurement
    _units.code                     angstroms
     save_
```

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

references