International Tables for Crystallography (2006). Vol. G, Chapter 2.1, pp. 13–19.

2.1. Specification of the STAR File

BY S. R. HALL AND N. SPADACCINI

2.1.1. Introduction

A human language, in all its forms, is spoken, written and comprehended according to grammatical principles that evolve continuously with the need to express efficiently new ideas and experiences. In this chapter we will describe how similar principles have been developed to construct, describe and understand scientific data, such as numbers and codified text. The efficient and flexible expression of data may be achieved using grammatical rules similar to those of spoken languages, though the precise and unambiguous rendition and communication of data must preclude those nuances, subtleties and individual interpretation so important to the spoken and graphical world of poetry, literature and art. It is important to record these aspects of human endeavour, but they are distinctly different from the objectives of science, where information must be expressed with maximum precision and efficiency.

Understanding any language involves two fundamental steps – identification of the individual elements of a language sequence, and comprehension of the sequence structure. In a spoken language these steps normally comprise the simultaneous recognition of individual words and the understanding of their grammatical context. Such a simple decoding process belies the enormous potential for complexity in spoken languages. Nevertheless, most 'word sequences' are understood in this way and a similar approach may be used with non-textual data.

As discussed in Section 1.1.3, until quite recently most approaches to storing scientific data electronically were based on fixed-format structures. These are simple to construct and easy to comprehend provided the data layout is fixed and widely understood. That is, items, as well as lists of items, are written in a fixed sequential order that is mutually agreed on by those writing and reading the data. However, the preordained nature of a fixed format, which intentionally prevents changes to the data structure in a file, also poses a serious limitation for many scientific applications. This is because the nature of data used in scientific disciplines, such as in crystallography, evolves continuously and requires recording processes that are extensible and adaptable to change.

A lack of ready extensibility in the expression of data is particularly problematical for long-term archiving. For example, the recovery of information stored in a fixed format may be impossible if the layout details are lost or altered with time. Less rigid fixed-format variants, using keywords to identify groups of data, can improve the flexibility of data recording but they often preclude the introduction of new kinds of data.

In the 1980s it was widely recognized across the sciences that more general, extensible and expressive approaches were needed for recording, transmitting and archiving electronic data. This led to the development of free-format file structures. These file structures are the basis for universal file formats intended to: (*a*) store all kinds of data; (*b*) be independent of computer hardware (*i.e.* portable); (*c*) be both machine-parsable and human-understandable; (*d*) adapt to future data evolution (*i.e.* be extensible and robust); and (*e*) facilitate data structures of any complexity (*i.e.* have rich syntax capabilities).

The extent to which these objectives can be met determines the universality of a data-storage approach. Several of these properties are difficult to achieve simultaneously, and the compromises that have been adopted have largely determined the success of universal data languages in different fields. The STAR File is a universal data language applicable to all scientific disciplines, and the Crystallographic Information File described in Chapter 2.2 of this volume is a specific instance of the STAR format that has been adopted by the crystallographic community.

2.1.2. Universal data language concepts

As discussed above, the basic precepts of a universal data language parallel those of spoken languages in that a syntax and a grammar are used to describe and arrange (*i.e.* present) information, and hence define its comprehension. In particular, because data *values* (as numbers, symbols or text) are not often self-descriptive (*e.g.* a temperature value as a number alone cannot be distinguished from a number representing an atomic weight) interpretation depends critically on appropriate cues to the meaning and organization of data. A cue, in this context, has traditionally been prior knowledge of a fixed format, but can be explicit descriptions of data items, or even embedded codes identifying the relationship between data items. Cues represent the contextual thread of a data language, in the same way that grammatical rules provide words in natural-language sentences with meaning, and convert computer languages into executable code. This section will consider the concepts of data and languages that are important in understanding the syntax and purpose of a STAR File.

The first requirement of a language for electronic data transmission is that it be computer-interpretable, *i.e.* parsable. One should be able to read, interpret, manipulate and validate data entirely by machine. The other essential attributes of a universal file, as listed in Section 2.1.1, are flexibility, extensibility and applicability to all kinds of data. In modern-day science, data items change and expand continually and it is therefore paramount that a data language can accommodate new kinds of data seamlessly. It is the inflexibility and restrictive scope of fixed formats that limit their usefulness to crystallography for purposes other than local and temporary data retention.

2.1.2.1. Data models

Various approaches exist for meeting the objectives for a universal file listed in Section 2.1.1. The task is complicated by the fact that properties such as generality and simplicity, or accessibility and flexibility, are in many respects technically incongruent, and, depending on the nature of the data to be represented, particular compromises may be taken for efficiency or expediency. In other words, data languages, like spoken languages, are not equally efficient at expressing concepts, and the syntax of a language may need to be tailored to the target applications.

Affiliations: SYDNEY R. HALL, School of Biomedical and Chemical Sciences, University of Western Australia, Crawley, Perth, WA 6009, Australia; NICK SPADACCINI, School of Computer Science and Software Engineering, University of Western Australia, 35 Stirling Highway, Crawley, Perth, WA 6009, Australia.

This was one of the issues confronting the Working Party on Crystallographic Information formed by the IUCr in 1988 (Section 1.1.6) to decide on the most appropriate universal file data language for crystallography from those under development (McCarthy, 1990). It is interesting historically to note that one of these was HGML – not the web markup language we know today, but the *Human Genome Mapping Library* language. Another language considered by the working party was ASN.1 (ISO, 2002) used by the National Institute of Standards and Technology and several US Government departments. It is an accepted ANSI and ISO standard for data communication and is supported by software, such as NIST's OSI Toolkit. ASN.1 possesses a rich set of language constructs suited to representing complex data, but suffers from data identifiers that are encoded and not human-readable, and a syntax that is verbose (particularly for repetitive data items such as those common in crystallography). These characteristics mean that a typical Protein Data Bank (PDB) file expressed in ASN.1 notation increases in size by up to a factor of 5. This was hardly an attraction in the 1980s when storage media were very expensive. Moreover, the ASN.1 syntax is not particularly intuitive, and is difficult to read and to construct. In contrast, the STAR File proposed at the first WPCI meeting had a relatively simple syntax, was human-readable and provided a concise structure for repetitive data. It also proved suitable for constructing electronic dictionaries, as will be discussed in later chapters. However, its serious and well recognized weakness in 1988 was that any recording approach using a simple syntax to encode complex data must involve sophisticated parsing software, and at that time only the prototype software (*QUASAR*; Hall & Sievers, 1990) was available. It was therefore not a straightforward decision for the WPCI to decide to recommend the STAR File syntax as a more appropriate data language for crystallographic applications. It was this decision that led to the development of the CIF approaches described in this volume.

2.1.3. The syntax of the STAR File

The syntax of the STAR File (Hall, 1991; Hall & Spadaccini, 1994) has been used to develop a number of discipline-specific exchange and archival approaches, including the Crystallographic Information File (CIF) (Hall *et al*., 1991), the Molecular Information File (MIF) (Allen *et al*., 1995), the dictionary definition language (DDL1) (Hall & Cook, 1995), the macromolecular dictionary definition language (DDL2) (Westbrook & Hall, 1995) and the STAR dictionary definition language (StarDDL) (Spadaccini *et al*., 2000). The details of the CIF, MIF, DDL1 and DDL2 approaches are given in Chapters 2.2, 2.4, 2.5 and 2.6, respectively.

A STAR File is a sequential file containing lines of standard ASCII characters. A file may be divided into any number of discrete sets of unique data items. Sets may be in the form of data blocks, global blocks or save frames. The syntax rules for these sets are given below in descriptive form. A more rigorous description of the STAR File syntax is given in Appendix 2.1.1 in extended Backus–Naur form (McLennon, 1983).

The STAR File is a free-form language in which spaces (ASCII 32), vertical tabs (ASCII 11) and horizontal tabs (ASCII 9) are collectively referred to as <blank>, and newlines (ASCII 10), form feeds (ASCII 12) and carriage returns (ASCII 13) are collectively referred to as <terminate>. White spaces <wspace>, used to separate lexical tokens within the file, are all characters in the joined set of <blank> and <terminate>.

2.1.3.1. Text string

A text string is defined as any of the following.

(*a*) A sequence of non-white-space characters on a single line excluding a leading underscore <_> (ASCII 95).

Examples:

```
5.324
light-blue
```

(*b*) A sequence of characters on a single line containing the leading digraph <wspace><'> and the trailing digraph <'><wspace>. <'> is a single-quote character (ASCII 39) and <wspace> is white space.

Examples:

```
'light blue'
'classed as "unknown"'
'Patrick O'Connor'
```

Note that the use of the <'> character in the text string that is bounded by a <'> character is not precluded unless it is immediately followed by <wspace>. The leading and trailing digraphs serve to delimit the string and do not form part of the data. In the above example the value associated with the text field 'light blue' is **light blue**.

(*c*) A sequence of characters on a single line containing the leading digraph <wspace><"> and the trailing digraph <"><wspace>. <"> is a double-quote (ASCII 34) character and <wspace> is white space.

Examples:

```
"low melting point"
"Patrick O'Connor"
"Doug Collins' crystal"
"classed as "unknown""
```

The use of the <"> character in the text string that is bounded by a <"> character is not precluded unless it is immediately followed by <wspace>. The leading and trailing digraphs serve to delimit the string and do not form part of the data.

The text strings of type (*a*), (*b*) and (*c*) cannot contain line-breaking characters, and therefore cannot span multiple lines. There can be more than one text string per line if each value is preceded by a data name, or the values are part of a looped list (see Section 2.1.3.5).

(*d*) A sequence of lines starting with <terminate><;> and finishing with <terminate><;>, where <;> is the semicolon character (ASCII 59).

Example:

```
; School of CSSE
  UWA
;
```

The requirement that the <;> character be the first character on the line does not preclude the presence of other characters on the same line, in as much as it does not violate the STAR File structure.

The leading and trailing digraphs delimit the text field and do not form part of the data. The character sequence between the digraphs, including any line-breaking characters, constitutes the value of the text field. The value associated with the above example is <blank>**School**<blank>**of**<blank>**CSSE** <terminate><blank><blank>**UWA** (note in particular that the <terminate> character preceding the final ; delimiter is *not* part of the value).

2.1.3.2. Data name

A data name (or tag) is the identifier of a data value (see Section 2.1.3.3) and is a sequence of non-white-space characters starting with an underscore character <_> (ASCII 95).

Example:

```
_publication_author_address
```

2.1.3.3. Data value

A data value is a text string preceded by its identifying data name. Privileged keywords, such as described in Sections 2.1.3.5 to 2.1.3.8, are excluded from this definition.

2.1.3.4. Data item

A data item is a data value and its associated data name. Each data item stored in a STAR File is specified with this combination.

2.1.3.5. Data loop list

A looped list consists of the keyword `loop_` followed by

(*a*) a sequence of data names (possibly with nested `loop_` constructs); and

(*b*) a sequence of loop packets, each containing data values which are identified in the same order as the data names in (*a*).

A looped list specifies a table of data in which the data names represent the 'header descriptors' for columns of data and the packets represent the rows in the table. Looped lists may be nested to any level. Each loop level is initialized with the `loop_` keyword and is followed by the names of data items in this level. Data values that follow the nested data declarations must be in exact multiples of the number of data names. Each loop level must be terminated with a `stop_`, except the outermost (level 1) which is terminated by either a new data item or the privileged strings indicating a save frame (Section 2.1.3.6), a data block (Section 2.1.3.7), a global block (Section 2.1.3.8) or an end of file.

An example of a simple one-level loop structure is:

```
loop_
    _atom_identity_number
    _atom_type_symbol       1 C   2 C    3 O
```

Nested (multi-level) looped lists contain matching data packets [as per (*b*) above] and an additional `stop_` to terminate each level of data. Here is a simple example of a two-level nested list.

```
loop_
    _atom_id_number
    _atom_type_symbol
    loop_
        _atom_bond_id_1
        _atom_bond_id_2
        _atom_bond_order
             1 C     1 2 single  1 3 double stop_
             2 C     2 1 single             stop_
             3 O     3 1 double             stop_
```

The matching of data names to value packets is applied at each loop level. Initially the data values are matched to the data names listed in the outermost level loop. This process is iterated to successively inner levels. At the innermost loop level, data matching is maintained until a `stop_` is encountered. This returns the matching process to the next outer level. The matching process is recursive until the loop structure is depleted. Here is an example of a three-level loop structure.

```
loop_
    _atomic_name
     loop_
          _level_scheme
          _level_energy
           loop_
                _function_exponent
                _function_coefficient
 hydrogen
        (2)->[2]    -0.485813
        1.3324838E+01    1.0
        2.0152720E-01    1.0 stop_
        (2)->[2]    -0.485813
        1.3326990E+01    1.0
        2.0154600E-01    1.0 stop_
        (2)->[1]    -0.485813
        1.3324800E-01    2.7440850E-01
        2.0152870E-01    8.2122540E-01 stop_
        (3)->[2]    -0.496979
        4.5018000E+00    1.5628500E-01
        6.8144400E-01    9.0469100E-01
        1.5139800E-01    1.0000000E+01 stop_ stop_
```

2.1.3.6. Save frame

A save frame is a set of unique data items wholly contained within a data block. The frame starts with a `save_`*framecode* statement, where the *framecode* is a unique identifying code within the data block. Each frame is closed with a `save_` statement.

Example:

```
data_example
save_phenyl
    _object_class    molecular_fragment
     loop_
          _atom_identity_node
          _atom_identity_symbol 1 C 2 C 3 C 4 C 5 C 6 C
save_
loop_ _molecular_fragments   $ethyl  $phenyl $methyl
```

A save frame has the following attributes:

(*a*) A save frame may contain data items and loop structures but not other save frames [see (*f*)].

(*b*) The scope of the data specified in a save frame is the save frame in which it is specified.

(*c*) Data values in a save frame are distinct from any identical items in the parent data block.

(*d*) A save frame may be referenced within the data block in which it is specified using a data item with a value of `$`*framecode*.

Example:

```
loop_ _amino_acid_seq
      _amino_acid_data   1 $tyr 2 $arg 3 $arg 4 $leu
```

where 'arg', 'tyr' and 'leu' are frame codes identifying three save frames of data.

(*e*) A frame code must be unique within a data block.

(*f*) A save frame may not contain another save frame, but it may contain references to other save frames in the same data block using frame codes.

2.1.3.7. Data block

A data block is a set of data containing any number of unique items and save frames. A data block begins with a `data_`*blockcode* statement, where *blockcode* is a unique identifying name within a file. A data block is closed by another `data_`*blockcode* statement, a `global_` statement or an end of file.

Example:

```
data_rhinovirus
all information relevant to rhinovirus included here

data_influenza
all information relevant to influenza virus
included here
```

A data block has the following attributes:

(*a*) A block code must be unique within the file containing the data block.

(*b*) Data blocks may not be referenced from within a file [in contrast to save frames – see Section 2.1.3.6(*d*)].

(*c*) The scope of data specified in a data block is the data block. The value of a data item is always associated with the data block in which it is specified.

(*d*) Data specifications in a data block are unique, except they may be repeated within a save frame. Data specifications in a save frame are independent of the parent data block specifications.

(*e*) If a data item is not specified in a given data block, the global value is assumed. If a global value is not specified, the value is unknown.

2.1.3.8. Global block

A global block is a set of data items which are implied to be present in all data blocks which follow in a file, unless specified explicitly within a data block. A global block starts with a `global_` keyword and is closed by a `data_`*`blockcode`* statement or an end of file.

Example:

```
global_
information that is default within subsequent
data blocks
data_influenza
```

A global block has the following attributes:

(*a*) The scope of global data is from the point of declaration to the end of file.

(*b*) A global block may contain data items, loop structures and save frames.

(*c*) Multiple global blocks are concatenated to form a single block in which the last item specification has precedence.

(*d*) A data item specified within a data block has precedence over a data item specified in a prior global block.

2.1.3.9. Data sets and scopes

A data set is the generic term for a unique set of data. A STAR File may contain three types of data sets: global blocks, data blocks and save frames. The attributes of data sets are as follows.

(*a*) A file may contain any number of data sets.

(*b*) The data names defined within a data set must be unique to that set. That is, all `data_`*`blockcode`* names must be unique within the file, all data names must be unique within a `global_` block, all data names and `save_`*`framecode`*s must be unique within a data block, and all data names must be unique within a save frame.

(*c*) The scope of data sets is hierarchical (Fig. 2.1.3.1). Global blocks encompass all following data blocks; data blocks scope all contained save frames.

(*d*) The scope of a save frame is all data items contained within the frame.

(*e*) The scope of a data block is the boundaries of the data block, *i.e.* the end of the file or the start of the next data block, including any contained save frames. The same data item may be defined within a save frame and within the parent data block. All specifications of this item will be recognized when accessing the data block.

(*f*) The scope of a global block is the file, from the point of invocation to the end of file or the start of the next global block. It encompasses all contained global data items, data blocks and save frames. Globally specified data are active provided identical items are not specified in subsequent data sets.

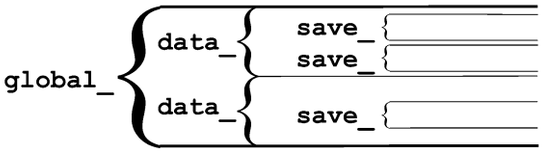

Fig. 2.1.3.1. Nested scopes of STAR data sets.

2.1.3.10. Privileged constructs

The following constructs are privileged.

(*a*) Text strings starting with the character sequences `data_`, `loop_`, `global_`, `save_` or `stop_` are privileged words (keywords) and may not be used as values in text strings of the type defined in Section 2.1.3.1(*a*).

(*b*) A sharp character `<#>` (ASCII 35) is an explicit end-of-line signal provided it is not contained within a text string of the types defined in Section 2.1.3.1(*b*), (*c*) or (*d*). Characters on the same line and following an active sharp character are considered as comment text.

2.1.3.11. Using stop_ in looped lists

In Section 2.1.3.5 we discuss how `stop_` is used to terminate a loop of data values and to return the looped list to the next outer nesting level. This same construction applies in the looped list of data names. The following, although not particularly intuitive, is a valid construction.

```
loop_
    _atom_id_number
        loop_
            _atom_bond_id_1
            _atom_bond_id_2
            _atom_bond_order  stop_
    _atom_type_symbol
                  1  1 2 single  1 3 double stop_ C
                  2  2 1 single             stop_ C
                  3  3 1 double             stop_ O
```

This is equivalent to the loop definition given in Section 2.1.3.5. One can use the `stop_` in name definitions to inhibit the nesting of loops in the definitions.

Appendix 2.1.1
Backus–Naur form of the STAR syntax and grammar

This description of the STAR syntax and grammar is annotated to clarify issues that cannot be represented in a pure extended Backus–Naur form (EBNF) definition.

The allowed character set in STAR is restricted to ASCII 09–13, 32–126. Other characters from the ASCII set are illegal. If such characters are present in a file the error state is well defined, but the functionality of the error handler is not specified. For instance, one may choose to return an illegal file exception and terminate the application or equally one may choose to ignore and skip over the illegal characters.

The concept of white space `<wspace>` includes a comment, since these only serve (in a parser sense) to delimit tokens anyway. We adopt the convention here of enclosing terminal symbols in single forward quotes. There are necessary provisos to this, and for representing formatting characters they are:

'\'' represents the single-quote character, *i.e.* the \ is an escape character.

'\\' represents the single backslash character, *i.e.* the \ is an escape character.

'\f' represents the form-feed character, *i.e.* ASCII 12.

'\n' represents the new-line character, *i.e.* ASCII 10.

'\r' represents the carriage-return character, *i.e.* ASCII 13.

'\t' represents the tab character, *i.e.* ASCII 09.

'\v' represents the vertical-tab character, *i.e.* ASCII 11.

There are STAR specifications not definable in the EBNF. The EBNF can be used to define the tokenization of the input stream, and a STAR parser should test that the following condition is true. The number of data elements in <data_loop_values> of a <data_loop> production *must* be an integer multiple of the number of data names in the associated <data_loop_field>.

The STAR syntax specified in the EBNF follows.

A2.1.1.1. Lexical tokens

We accept a space, a horizontal and a vertical tab as <blank>.

```
<blank> ::= ' '| '\t' | '\v'  (ASCII 32 09 11)
```

The non-printing single ASCII characters 10, 12, 13 or any sequence of these are always interpreted as being a single line terminator. In this way there should not be operating-system-dependent ambiguity in those architectures that use character sequences as line terminators. This necessarily requires that these characters can only be used for line termination.

```
<terminate> ::= { '\n' | '\r' | '\f' }+
                (ASCII 10 13 12)
```

We define a 'comment' to be initiated with <blank> or <terminate> and the character #, followed by any sequence of characters (which include <blank>). The only characters not allowed are those in the production <terminate>, and hence these characters terminate a comment. Note the requirement of a leading <blank> or <terminate> is dropped if the # character is the first character in the file.

```
<comment> ::= { <blank> | <terminate> }+ '#' <char>*
```

We accept as white space *all* elements in the above three productions. White spaces are the lexemes able to delimit the lexical tokens. Note that a comment is a legitimate white space because it must end with a line terminator, and hence delimits tokens.

```
<wspace> ::= { <blank> | <terminate> | <comment> }+
```

Non-blank characters are composed of all the characters in our set, excluding <blank> and <terminate> characters.

```
<non_blank_char> ::= <ordinary_char> | '"' | '#' | '$'
                   | '\'' | ';' | '_' | '[' | ']'
```

<char> characters are composed of all the characters in our set, excluding <terminate> characters.

```
<char> ::= <blank> |  <non_blank_char>
```

We define a 'line of text' to be a line contained within a semicolon-bounded text block. Hence the first character *cannot* be a semicolon, and is followed by any number of characters from the set <char> and terminated with a line-termination character or just the termination character. This allows for 'blank' lines in the semicolon-bounded text block.

```
<line_of_text> ::=
     { <not_a_semi_colon> <char>* <terminate>
     | <terminate> }
```

Productions for specific characters.

```
<D_quote>       ::= '"'  (ASCII 34)
<S_quote>       ::= '\'' (ASCII 39)
<semi_colon>    ::= ';'  (ASCII 59)
```

All printable characters *except* the double quote.

```
<not_a_D_quote> ::= <ordinary_char> | '#' | '$' | '\''
                  | ';' | '_' | '[' | ']' | <blank>
```

All printable characters *except* the single quote.

```
<not_an_S_quote> ::= <ordinary_char> | '"' | '#' | '$'
                   | ';' | '_' | '[' | ']' | <blank>
```

All printable characters *except* the left and right square brackets.

```
<not_an_S_bracket> ::= <ordinary_char> | '"' | '#'
                     | '$' | ';' | '_'   | '\''
                     | <blank>
```

All printable characters *except* the semicolon.

```
<not_a_semi_colon> ::= <ordinary_char> | '"' | '#'
                     | '$' | '\'' | '_' | '[' | ']'
                     | <blank>
```

Ordinary characters are all those printable characters that can initiate a non-quoted text string. These exclude the special characters, ", #, $, ' and _ and in some cases ;.

```
<ordinary_char> ::=
       '!' | '%' | '&' | '(' | ')' | '*' | '+' | ','
     | '-' | '.' | '/' | '0' | '1' | '2' | '3' | '4'
     | '5' | '6' | '7' | '8' | '9' | ':' | '<' | '='
     | '>' | '?' | '@' | 'A' | 'B' | 'C' | 'D' | 'E'
     | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'
     | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
     | 'V' | 'W' | 'X' | 'Y' | 'Z' | '\\' | '^' | '`'
     | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
     | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
     | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x'
     | 'y' | 'z' | '{' | '|' | '}' | '~' | <blank>
```

The keywords (in a case-insensitive form).

```
<DATA_> ::= {'d'|'D'} {'a'|'A'} {'t'|'T'} {'a'|'A'} '_'
<LOOP_> ::= {'l'|'L'} {'o'|'O'} {'o'|'O'} {'p'|'P'} '_'
<GLOBAL_> ::= {'g'|'G'} {'l'|'L'} {'o'|'O'} {'b'|'B'}
              {'a'|'A'} {'l'|'L'} '_'
<STOP_> ::= {'s'|'S'} {'t'|'T'} {'o'|'O'} {'p'|'P'} '_'
<SAVE_> ::= {'s'|'S'} {'a'|'A'} {'v'|'V'} {'e'|'E'} '_'
```

The operating-system-dependent end-of-file marker.

```
<EOF>   ::= end-of-file marker
```

A2.1.1.2. STAR grammar

A STAR File may be an empty file, or it may contain one or more data blocks or global blocks.

```
<STAR_File> ::=
     <wspace>* {<data_block> | <global_block>}*
```

There can be any amount of white spaces (remember <wspace> includes comments) before and at least one white space or an end of file (EOF) after a data or global block. This forces white space between data (and global) blocks in a single file. There must be *at least* one data item in any data or global block. This means a file consisting of just a data or global block heading is *invalid*.

```
<data_block>   ::= <data_heading>
                   {<data> | <save_block>}+
                   { <wspace>+ | <EOF> }
<global_block> ::= <GLOBAL_> {<data> | <save_block>}+
                   { <wspace>+ | <EOF> }
```

There can be any amount of white spaces (remember <wspace> includes comments) before a save-frame block. This forces white space between save-frame blocks also. There is no need to include the { <wspace>+ | <EOF> } found in data and global blocks, since those productions cover the situation of a save-frame block terminating the file.

```
<save_block> ::= <wspace>+ <save_heading>
                 <data>+ <wspace>+ <SAVE_>
```

A data-block or save-frame heading consists of the relevant five-character keyword (case-insensitive) immediately followed by at least one non-blank character. This does not preclude the associated block name or frame name consisting of just one or more punctuation characters.

```
<data_heading> ::= <DATA_> <non_blank_char>+
<save_heading> ::= <SAVE_> <non_blank_char>+
```

Data come in the following three forms.

(1) A data-name tag separated from its associated value by a trailing <blank>. Note it is explicitly a <blank> and not a <wspace>. These are *type I* data.

(2) A data-name tag separated from its associated value by a <terminate>. These are *type II* data.

(3) Looped data.

```
<data> ::= { <wspace>+ <data_name> <wspace>* <blank>
             <type_I_data_value> }
           | { <wspace>+ <data_name> <wspace>*
               <terminate> <type_II_data_value> }
           | <data_loop>
```

We must allow for white space preceding the loop_ (case-insensitive) keyword, since this is not covered by any of the other productions.

```
<data_loop> ::= <wspace>+ <LOOP_> <data_loop_field>
                <data_loop_values>
```

The name list for a loop must include at least one data name or a nested loop.

```
<data_loop_field> ::=
                  { <wspace>+ <data_name>
                  | <wspace>+ <LOOP_> <data_loop_field>
                    [<wspace>+ <STOP_>] }+
```

A data name is initiated by an underscore character and followed by one or more non-blank and non-terminating characters from the STAR character set. This does not preclude data names consisting of just one or more punctuation characters.

```
<data_name> ::= '_' <non_blank_char>+
```

Loop values are represented in the same way as the <data> production, except that the possibility of nested data loops introduces the need for the stop_ keyword.

```
data_loop_values> ::=
     { { <wspace>* <blank> <type_I_data_value> }
     | { <wspace>* <terminate> <type_II_data_value> }
     | <wspace>+ <STOP_> }+
```

Data values of type I data are immediately preceded by a <blank>. Data values of type II data are immediately preceded by a <terminate>.

```
<type_I_data_value> ::= <non_quoted_I_string>
     | { '"' <D_quote_string> '"' }
     | { '\'' <S_quote_string> '\'' }
     | { '[' <SB_bounded_string> ']' }

<type_II_data_value> ::= <non_quoted_II_string>
     | { '"' <D_quote_string> '"' }
     | { '\'' <S_quote_string> '\'' }
     | { ';' <SC_bounded_string> ';' }
     | { '[' <SB_bounded_string> ']' }
```

A type-I unquoted string is immediately preceded by a <blank>. It cannot begin with a number of characters (the complement of the <ordinary_char> set) *i.e.* ", #, $, ', [,] and _. However, it can begin with a semicolon. Then it is followed by any number of non-blank characters.

```
<non_quoted_I_string> ::=
     { <ordinary_char> | <semi_colon> }
     <non_blank_char>*
```

A type-II unquoted string is immediately preceded by a line break. As with type I, it too cannot begin with a ", #, $, ', [,] or _. It also *cannot* begin with a semicolon, since this would match the semicolon-delimited data production.

```
<non_quoted_II_string> ::=
     <ordinary_char> <non_blank_char>*
```

Specific exceptions to lexemes which match both types of unquoted strings are:

(1) No string beginning with an underscore is an unquoted string.

(2) No string that matches a production for <data_heading>, <save_heading>, <LOOP_>, <STOP_>, <SAVE_> or <GLOBAL_> is an unquoted string.

If one wishes to define data values which match lexemes excluded in cases (1) and (2) above, they should be *quoted* data values.

The string between a set of double quotes can consist of any character that is not a double quote, or it can be a double quote as long as it is immediately followed by a non-blank character or any number of double quotes at the end of the string. This final rule picks up cases of double-quote delimited strings that end in one or more double quotes, like "ABC"".

```
<D_quote_string> ::= { '"' <non_blank_char>
                     | <not_a_D_quote> }* { '"' }*
```

The string between a set of single quotes can consist of any character that is not a single quote, or it can be a single quote as long as it is immediately followed by a non-blank character or any number of single quotes at the end of the string. This final rule picks up cases of single-quote delimited strings that end in one or more single quotes, like 'ABC''.

```
<S_quote_string> ::= { '\'' <non_blank_char>
                     | <not_an_S_quote>}* { '\'' } *
```

The string bounded by semicolons can begin with any number of characters (including those in the <blank> production) but is necessarily terminated by a line break. This forces a line break on the line that contains the 'opening' semicolon. After the first line, one can have any number of <line_of_text>. Note we treat the first line as special, since it can contain a leading semicolon, which is not true of <line_of_text>. A <line_of_text> is always terminated with a line break, thus ensuring the closing semicolon is in column 1.

```
<SC_bounded_string> ::=
     <char>* <terminate> <line_of_text>*
```

The string bounded by square brackets can consist of any character including <terminate> and <blank>, and excluding the characters [and] unless they are escaped or are balanced.

```
<SB_bounded_string> ::= { <not_an_S_bracket>
                        | '\\' '[' | '\\' ']'
                        | <terminate>
                        | <SB_bounded_string> }*
```

The development of the STAR File, and particularly its application to crystallographic data, involved many people. The major contributions are acknowledged in the references below and other chapters in this volume. We name here only contributors who are not listed elsewhere in STAR-related publications. Particular thanks are due to Richard Goddard, Ted Maslen, Andre Authier, Philip Coppens, Jim King, Mike Dacombe, Peter Strickland, Mike Hoyland and George Sheldrick for their interest, support and commitment during the development of the STAR File concepts and its applications to crystallography.

References

Allen, F. H., Barnard, J. M., Cook, A. P. F. & Hall, S. R. (1995). *The Molecular Information File (MIF): core specifications of a new standard format for chemical data. J. Chem. Inf. Comput. Sci.* **35**, 412–427.

Hall, S. R. (1991). *The STAR File: a new format for electronic data transfer and archiving. J. Chem. Inf. Comput. Sci.* **31**, 326–333.

Hall, S. R., Allen, F. H. & Brown, I. D. (1991). *The Crystallographic Information File (CIF): a new standard archive file for crystallography. Acta Cryst.* A**47**, 655–685.

Hall, S. R. & Cook, A. P. F. (1995). *STAR dictionary definition language: initial specification. J. Chem. Inf. Comput. Sci.* **35**, 819–825.

Hall, S. R. & Sievers, R. (1990). *QUASAR: CIF syntax checking and file manipulation tool.* ftp://ftp.iucr.org/pub/quasar.

Hall, S. R. & Spadaccini, N. (1994). *The STAR File: detailed specifications. J. Chem. Inf. Comput. Sci.* **34**, 505–508.

ISO (2002). ISO/IEC 8824-1. *Abstract Syntax Notation One (ASN.1). Specification of basic notation.* Geneva: International Organization for Standardization.

McCarthy, J. L. (1990). *Data interchange standards for biotechnology: issues and alternatives.* National Center for Biotechnical Information Report. NIH/DOE.

McLennon, B. J. (1983). *Principles of programming languages, design, evaluation and implementation*. New York: Holt, Rinehart and Winston.

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

Westbrook, J. D. & Hall, S. R. (1995). *A Dictionary Description Language for Macromolecular Structure.* http://ndbserver.rutgers.edu/mmcif/ddl.