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.

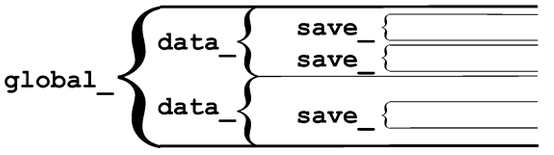

Fig. 2.1.3.1. Nested scopes of STAR data sets.

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.