International Tables for Crystallography (2006). Vol. G, Section 2.2.7.3, pp. 30–32.

the rest of the parsing may be done more efficiently. The convention followed in this document is to mark the 'non-terminal' tokens that are built up out of actual strings of characters or which do not have an immediate representation as printable characters by angle brackets, <>, and to indicate the tokens that are actual strings of characters as quoted strings of characters.

(40) The precise division between a lexical scan and a full parse is a matter of convenience. A suggested division is presented. Before getting to that point, however, there are some highly machine-dependent matters that need to be resolved. There must be a clear understanding of the character set to be used, and of how files and lines begin and end. The character set will be specified in terms of printable characters and a few control characters from the 7-bit ASCII character set. In addition, we will need some means of specifying the end of a line.

(41) The character set in CIF is restricted to the ASCII control characters `<HT>` (horizontal tab, position 09 in the ASCII character set), `<NL>` (newline, position 10 in the ASCII character set, also named `<LF>`) and `<CR>` (carriage return, position 13 in the ASCII character set), and the printable characters in positions 32–126 of the ASCII character set. These are the characters permitted by STAR with the exception of `VT` (vertical tab, position 11 in the ASCII character set) and `FF` (form feed, position 12 in the ASCII character set). In general it is poor practice to use characters that are not common to all national variants of the ISO character set. On systems or in programming languages that do not 'work in ASCII', the characters themselves may have different numeric values and in some cases there is no access to all the control characters.

(42) The `<eol>` token stands for the system-dependent end of line.

Implementation note: CIF implementations may follow common HTML and XML practice in handling `<eol>`:

> '[On many modern systems,] lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA). To simplify the tasks of applications, the characters passed to an application . . . must be as if the . . . [parser] normalized all line breaks in external parsed entities . . . on input, before parsing, [*e.g.*] by translating both the two-character sequence #xD #xA and any #xD that is not followed by #xA to a single #xA character.'

(From the XML specification http://www.w3.org/TR/2000/REC-xml-20001006.)

Because Unix systems use `\n` (the ASCII LF control character, or #xA), MS Windows systems use `\r\n` (the ASCII CR control character, or #xD, followed by the ASCII LF control character, or #xA) and classic MacOS systems use `\r`, a parser which covers a wide range of systems in a reasonable manner could be constructed using a pseudo-production for `<eol>` such as

```
<eol> ::= { <LF> | <CR><LF> | <CR> }
```

provided the supporting infrastructure (such as the lexer) deals with the necessary minor adjustment to ensure that each end of line is recognized and that all end-of-line control characters are filtered out from the portions of the text stream that are to be processed by other productions. One case to handle with care is the end-of-document case. It is not uncommon to encounter a last line in a document that is not terminated by any of the above-mentioned control characters. Instead, it may be terminated by the end of the character stream or by a special end-of-text-document control character [*e.g.* #x4 (control-D) or #x1A (control-Z)]. A CIF parser should normalize such unterminated terminal lines to appear to an application as if they had been properly terminated. On the other hand, care should also be taken so that in multiple generations of CIF processing such processing does not result in an ever-growing 'tail' of empty lines at the end of a CIF document.

This discussion is *not* meant to imply that a parser for a system that uses one of these line-termination conventions must recognize a CIF written using another of these line-termination conventions.

This discussion is *not* meant to imply that parsers on systems that use other line-termination conventions and/or non-ASCII character sets need to handle these ASCII control characters.

In processing a valid CIF document, it is always sufficient that a parser be able to recognize the line-termination conventions of text files local to its system environment, and that it be able to recognize the local translations of `<SP><HT>` and the printable characters used to construct a CIF.

However, when circumstances permit, if a parser is able to recognize 'alien' line terminations, it is permissible for the parser to accept and process the CIF in that form without treating it as an error.

In writing CIF documents, the software that emits lines should follow the text-file line-termination conventions of the target system for which it is writing the CIF documents, and not mix conventions from multiple systems. In transmitting a CIF document from system to system, software should be used that causes the document to conform to the line-termination conventions of the target system. In most cases this objective can best be achieved by using 'text' or 'ascii' transmission modes, rather than 'binary' or 'image' transmission modes.

(43) In order to write the grammar, we need a way to refer to the single-quote characters which we use both to quote within the syntax and to quote within a CIF. To avoid system-dependent confusion, we define the following special tokens:

Token	Meaning
`<SP>`	' ', the printable space character
`<HT>`	the horizontal-tab character on the system
`<eol>`	the machine-dependent end of line
`<noteol>`	the complement of the above; any character that does not indicate the machine-dependent end of line
`<single_quote>`	the apostrophe, '
`<double_quote>`	the double-quote character, "

(44) There are CIF specifications not definable directly in a context-free Backus–Naur form (BNF). Restrictions in record and data-name lengths, and the parsing of text fields and quoted character strings are best handled in the initial lexical scan. A pure BNF can then be used to parse the tokenized input stream.

2.2.7.3. Lexical tokens

(45) We define a 'comment' to be initiated with the character `#`. This can be followed by any sequence of characters (which include `<SP>` or `<HT>`). The only characters not allowed are those in the production `<eol>`, which `<eol>` terminates a comment. A comment is recognized only at the beginning of a line or after blanks, *i.e.* only after space, tab or `<eol>`. For this reason we define both comments and 'tokenized comments'. No portion of the essential machine-readable content within a CIF is conveyed by the comments. Comments are for the convenience of human readers of CIFs and may be freely introduced or removed. Note however the optional structured comment sanctioned in paragraph (34) above, which has the purpose of indicating the file type and revision level to general-purpose file-handling software.

```
<Comments>          ::= { '#' {<AnyPrintChar>}* <eol>}+
<TokenizedComments> ::= { <SP>|<HT>|<eol> }+ <Comments>
```

(46) We accept as white space all appropriate combinations of spaces, tabs, end of lines and comments, as well as the beginning of the file. White space are the characters able to delimit the lexical tokens.

```
<WhiteSpace> ::= {<SP>|<HT>|<eol>|<TokenizedComments>}+
```

(47) Non-blank characters are composed of all the characters in our set, excluding `<SP>` and `<HT>` and `<eol>` characters.

```
<NonBlankChar> ::= <ordinary_char>|<double_quote>|'#'|
     '$'|<single_quote>|'_'|';'|'['|']'
```

(48) `AnyPrintChar` characters are composed of all the characters in our set excluding `<eol>` characters.

```
<AnyPrintChar> ::= <ordinary_char>|<double_quote>|'#'|
     '$'|<single_quote>|'_'|<SP>|<HT>|';'|'['|']'
```

(49) We define a 'line of text' to be a line contained within a semicolon-bounded text field. Hence the first character *cannot* be a semicolon; it may be followed by any number of characters from the set `<char>` and terminated with a line-termination character. We define the characters in `<TextLeadChar>` as those in `<AnyPrintChar>` except for the semicolon.

```
<TextLeadChar> ::= <ordinary_char>|<double_quote>|'#'|
     '$'|<single_quote>|'_'|<SP>|<HT>|'['|']'
```

(50) Ordinary characters are all those printable characters that can initiate a non-quoted character string. These exclude the special characters `"`, `#`, `$`, `'`, `[`, `]` and _, and in some cases `;`.

```
<OrdinaryChar> ::=
       '!'|'%'|'&'|'('|')'|'*'|'+'|','|'-'|'.'|'/'|'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'|'{'|'|'
      |'}'|'~'
```

(51) The *reserved word* **`data_`** (in a case-insensitive form).

```
<DATA_> ::= {'d'|'D'} {'a'|'A'} {'t'|'T'} {'a'|'A'} '_'
```

(52) The *reserved word* **`loop_`** (in a case-insensitive form).

```
<LOOP_> ::= {'l'|'L'} {'o'|'O'} {'o'|'O'} {'p'|'P'} '_'
```

(53) The *reserved word* **`save_`** (in a case-insensitive form).

```
<SAVE_> ::= {'s'|'S'} {'a'|'A'} {'v'|'V'} {'e'|'E'} '_'
```

(54) The *reserved word* **`stop_`** (in a case-insensitive form).

```
<STOP_> ::= {'s'|'S'} {'t'|'T'} {'o'|'O'} {'p'|'P'} '_'
```

(55) The *reserved word* **`global_`** (in a case-insensitive form). This is actually a reserved word of STAR, but we define it here so that it may be explicitly excluded as an unquoted string. We do this so that any possible future adoption of STAR features will not invalidate existing CIFs.

```
<GLOBAL_> ::= {'g'|'G'} {'l'|'L'} {'o'|'O'} {'b'|'B'}
              {'a'|'A'} {'l'|'L'} '_'
```

(56) Quoted strings need to be recognized in the lexical scan, because their definition is context-sensitive. A string quoted by single quotes may contain a single quote as long as it is not followed by white space. A string quoted by double quotes may contain a double quote as long as it is not followed by white space. Formally we express this with context-sensitive productions. In practice, it requires a one-character look-ahead to decide to continue the scan if the opening quote is encountered, but the following character is not space, tab or end of line. When processing a semicolon-delimited text field, the column position has to be remembered to decide whether a semicolon should be recognized.

For a semicolon-delimited text string, failure to provide trailing white space is an error. The `<WhiteSpace>` on the left-hand side must evaluate to the same string instance on the right-hand side and the parse must terminate on the first valid match reading left to right.

```
<SingleQuotedString><WhiteSpace> ::=
     <single_quote>{<AnyPrintChar>}*<single_quote>
     <WhiteSpace>
<DoubleQuotedString><WhiteSpace> ::=
     <double_quote>{<AnyPrintChar>}*<double_quote>
     <WhiteSpace>
<TextField> ::= {<SemiColonTextField>}
<eol><SemiColonTextField> ::=
     <eol>';' { {<AnyPrintChar>}* <eol>
          {{<TextLeadChar> {<AnyPrintChar>}*}? <eol>}*
            } ';'
<BracketTextField> ::=
     '[' { <NonBracketChar>|<BracketTextField>}* ']'
```

(57) Tags and values are appropriate lexical tokens. The special values of '`.`' and '`?`' represent data that are inapplicable or unknown, respectively.

(i) No string that matches the production for `<LOOP_>` is accepted as a non-quoted string.

(ii) No string that matches the production for `<STOP_>` is accepted as a non-quoted string.

(iii) No string in which the initial five characters match the production for `<DATA_>` is accepted as a non-quoted string.

(iv) No string in which the initial five characters match the production for `<SAVE_>` is accepted as a non-quoted string.

(v) No string that matches the production for `<GLOBAL_>` is accepted as a non-quoted string.

Unquoted strings are described by a pair of productions to permit the initial letter of an unquoted string to be a semicolon so long as that does not occur at the beginning of a line. The parser is required to evaluate `<noteol>` to the same string instance on both sides of the production.

```
<Tag>       ::= '_'{ <NonBlankChar>}+
<Value>     ::= { '.'|'?'|<Numeric>|
                  <CharString>|<TextField> }
<Numeric>   ::= { <Number>|
                  <Number> '(' <UnsignedInteger> ')' }
<Number>    ::= { <Integer>|<Float> }
<Integer>   ::= { '+'|'-' }? <UnsignedInteger>
<Exponent> ::= { {'e'|'E' }|{'e'|'E' } { '+'|'- ' } }
                  <UnsignedInteger>
<UnsignedInteger> ::= {<Digit> }+
<Digit>     ::= {'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'}
<Float>     ::= { <Integer><Exponent>|
                  { {'+'|'-'} ?
                    { {<Digit>}* '.' <UnsignedInteger>} |
                      {<Digit>}+ '.' }
                    }
                    {<Exponent>} ?
                  }
                }
<CharString> ::= <UnquotedString>|<SingleQuotedString>|
     <DoubleQuotedString>
<eol><UnquotedString> ::=
     <eol><OrdinaryChar>{<NonBlankChar>}*
<noteol><UnquotedString> ::=
     <noteol>{<OrdinaryChar>|';'} {<NonBlankChar>}*
```

2.2.7.3.1. *CIF grammar*

(58) A CIF may be an empty file, or it may contain only comments or white space, or it may contain one or more data blocks. Comments before the first block are acceptable, and there must be white space between blocks.

```
<CIF> ::= <Comments>? <WhiteSpace>?
            { <DataBlock>
                { <WhiteSpace> <DataBlock> }*
                { <WhiteSpace> }?
            }?
```

(59) For a data block, there must be a data heading and zero or more data items or save frames.

```
<DataBlock> ::= <DataBlockHeading>
                  { <WhiteSpace>
                    { <DataItems> | <SaveFrame> }
                  }*
```

(60) A data-block heading consists of the five characters **data_** (case-insensitive) immediately followed by at least one non-blank character selected from the set of ordinary characters or the non-quote-mark, non-blank printable characters.

```
<DataBlockHeading> ::= <DATA_> { <NonBlankChar> }+
```

(61) For a save frame, there must be a save-frame heading, some data items and then the reserved word **save_**.

```
<SaveFrame> ::= <SaveFrameHeading>
                {<WhiteSpace> <DataItems>}+
                <WhiteSpace> <SAVE_>
```

(62) A save-frame heading consists of the five characters **save_** (case-insensitive) immediately followed by at least one non-blank character selected from the set of ordinary characters or the non-quote-mark, non-blank printable characters.

```
<SaveFrameHeading> ::= <SAVE_> { <NonBlankChar> }+
```

(63) Data come in two forms:

(i) A data-name tag separated from its associated value by a `<WhiteSpace>`.

(ii) Looped data. The number of values in the body must be a multiple of the number of tags in the header.

```
<DataItems>  ::= <Tag> <WhiteSpace> <Value> |
                 <LoopHeader> <LoopBody>
<LoopHeader> ::= <LOOP_> { <WhiteSpace> <Tag> }+
<LoopBody>   ::= <Value> { <WhiteSpace> <Value> }*
```

2.2.7.4. Common semantic features

2.2.7.4.1. *Introduction*

(1) The Crystallographic Information File (CIF) standard is an extensible mechanism for the archival and interchange of information in crystallography and related structural sciences. Ultimately CIF seeks to establish an ontology for machine-readable crystallographic information – that is, a collection of statements providing the relations between concepts and the logical rules for reasoning about them.

Essential components in the development of such an ontology are:

(*a*) the basic rules of grammar and syntax, described in Sections 2.2.7.1 to 2.2.7.3;

(*b*) a vocabulary of the tags or data names specifying particular objects;

(*c*) a taxonomy, or classification scheme relating the specified objects;

(*d*) descriptions of the attributes and relationships of individual and related objects.

In the CIF framework, the objects of discourse are described in so-called data dictionary files that provide the vocabulary and taxonomic elements. The dictionaries also contain information about the relationships and attributes of data items, and thus encapsulate most of the semantic content that is accessible to software. In practice, different dictionaries exist to service different domains of crystallography and a CIF that conforms to a specific dictionary must be interpreted in terms of the semantic information conveyed in that dictionary.

However, some common semantic features apply across all CIF applications, and the current document outlines the foundations upon which other dictionaries may build more elaborate taxonomies or informational models.

2.2.7.4.2. *Definition of terms*

(2) The definitions of Section 2.2.7.1.2 also hold for this part of the specification.

2.2.7.4.3. *Semantics of data items*

(3) While the STAR File syntax allows the identification and extraction of tags and associated values, the interpretation of the data thus extracted is application-dependent. In CIF applications, formal catalogues of standard data names and their associated attributes are maintained as external reference files called data dictionaries. These dictionary files share the same structure and syntax rules as data CIFs.

(4) At the current revision, two conventions (known as dictionary definition languages or DDLs) are supported for detailing the meaning and associated attributes of data names. These are known as DDL1 (Hall & Cook, 1995) and DDL2 (Westbrook & Hall, 1995), and they differ in the amount of detail they carry about data types, the relationships between specific data items and the large-scale classification of data items.

(5) While it may be formally possible to define the semantics of the data items in a given data file in both DDL1 and DDL2 data dictionaries, in practice different dictionaries are constructed to define the data names appropriate for particular crystallographic applications, and each such dictionary is written in DDL1 or DDL2 formalism according to which appears better able to describe the data model employed. There is thus in practice a bifurcation of CIF into two dialects according to the DDL used in composing the relevant dictionary file. However, the use of aliases may permit applications tuned to one dialect to import data constructed according to the other.

2.2.7.4.4. *Data-name semantics*

(6) Strictly, data names should be considered as void of semantic content – they are tags for locating associated values, and all information concerning the meaning of that value should be sought in an associated dictionary.

(7) However, it is customary to construct data names as a sequence of components elaborating the classification of the item within the logical structure of its associated dictionary. Hence a data name such as **_atom_site_fract_x** displays a hierarchical arrangement of components corresponding to membership of nested groupings of data elements. The choice of components readily indicates to a human reader that this data item refers to the fractional *x* coordinate of an atomic site within a crystal unit

references