2.2. Specification of the Crystallographic Information File (CIF)

By S. R. Hall and J. D. Westbrook
with Section 2.2.7 by S. R. Hall, N. Spadaccini, I. D. Brown, H. J. Bernstein, J. D. Westbrook and B. McMahon

2.2.1. Introduction

The term 'Crystallographic Information File' (CIF) refers to data and dictionary files conforming to the conventions adopted by the IUCr in 1990 and revised by the IUCr Committee for the Maintenance of the CIF Standard (COMCIFS). The CIF format is intended to meet the needs of a wide range of scientific applications within, and without, the discipline of crystallography. Parts 2 and 3 of this volume provide the full specification of the contents of CIF across the different crystallographic applications. The files used in these applications must conform to the same rules of syntax, and share certain properties and conventions in the way that information is presented. It is these common features that are discussed in this chapter.

The CIF family of applications uses a proper subset of the STAR File syntax described in Chapter 2.1. The STAR File grammar provides a very general approach to storing and accessing data values through the use of an associated data name, or tag. A CIF search tool, such as *Star_Base* (Chapter 5.2), can readily access a single data value, or set of values, using this tag without prior knowledge of the order of the file contents. It can also provide details of the context of the data within the file structure. Context, in this sense, is a fully annotated indication of the file structure in which the retrieved value was located. That is, whether it was located in a global declaration, a named save frame, or a looped list. In every case the data 'value' is simply a character string and the STAR File protocol itself imposes absolutely no meaning on that string. This leaves the interpretation of the value string (*e.g.* whether it is numerical or text) to the conventions of the applications used to read and write the STAR File.

The CIF approach to the permissive STAR File syntax is restrictive. In the first place, the earliest version of the CIF syntax (Hall *et al.*, 1991) did not adopt some of the grammatical (or syntactical) constructs available to the STAR File in order to facilitate existing crystallographic software approaches. This was in anticipation of likely short-term developments, and to encourage a rapid take-up of the CIF approach. For these reasons it was considered appropriate to adopt only data-block partitioning and a single, rather than multiple, level of looped lists. Save frames were adopted into the CIF later but only in CIF dictionary files written using the DDL2 dictionary definition language (see Chapter 2.6). It is relevant to point out, however, that the full STAR File syntax has been adopted for the storage of NMR experimental and structure data (Ulrich *et al.*, 1998).

In 2002, a COMCIFS review of the design and implementation of CIF led to a revised syntax specification, which was published in February 2003. This revised specification is reproduced in full in Section 2.2.7. It is important to note that after more than a decade of CIF usage, this revision contains few substantial changes to the design choices of the original version (Hall *et al.*, 1991). There have been some modest extensions to the lengths of data names and text lines, and a number of clarifications are introduced. In addition, various privileged labels used for STAR File constructs (*e.g.* global blocks, save frames and nested loops, as described in Chapter 2.1) have now been explicitly reserved (*i.e.* excluded from appearing in unquoted form in an existing CIF). This will allow the clean upward migration of future CIF syntax versions to the more complex data structures permitted in a STAR File, when and if these are later required by the community.

The remainder of this chapter is structured as follows. First, there is a brief description of CIF terminology (Section 2.2.2). This is followed by the syntax rules, corresponding to a subset of the STAR syntax, used by CIF data files (Section 2.2.3). The portability and archival issues that programmers must be aware of in applying CIF data in different computing environments are described in Section 2.2.4. They are also detailed in the formal specifications given at the end of the chapter. Section 2.2.5 describes the conventions regarding data typing and embedded semantics that are common to all CIF applications, and Section 2.2.6 outlines future possible ways of introducing metadata which would enable files to be linked to each other, and which would establish the nature of CIF contents within a more general framework of information storage systems. The final section of the chapter, Section 2.2.7, reproduces in full the formal specification documents approved by COMCIFS.

2.2.2. Terminology

A summary of the basic terminology used throughout this chapter and the volume follows. A more extensive description of this terminology is given as formal specifications in Section 2.2.7.1.2.

(i) A **CIF** is a file conforming to the specification presented in this chapter. The term includes both **data files** containing information on a structural experiment or its results (or similar scientific content) and the **dictionary files** that provide descriptions of the data identifiers used in such data files.

(ii) A **data name** or **tag** is an identifier (a string of characters beginning with an underscore character) of the content of an associated data value.

(iii) A **data value** is a string of characters representing a particular item of information. It may represent a single numerical value; a letter, word or phrase; extended discursive text; or in principle any coherent unit of data such as an image, audio clip or virtual-reality object.

(iv) A **data item** is a specific piece of information defined by a data name and its associated data value.

Affiliations: Sydney R. Hall, School of Biomedical and Chemical Sciences, University of Western Australia, Crawley, Perth, WA 6009, Australia; John D. Westbrook, Protein Data Bank, Research Collaboratory for Structural Bioinformatics, Rutgers, The State University of New Jersey, Department of Chemistry and Chemical Biology, 610 Taylor Road, Piscataway, NJ 08854-8087, USA; Nick Spadaccini, School of Computer Science and Software Engineering, University of Western Australia, 35 Stirling Highway, Crawley, Perth, WA 6009, Australia; I. David Brown, Brockhouse Institute for Materials Research, McMaster University, Hamilton, Ontario, Canada L8S 4M1; Herbert J. Bernstein, Department of Mathematics and Computer Science, Kramer Science Center, Dowling College, Idle Hour Blvd, Oakdale, NY 11769, USA; Brian McMahon, International Union of Crystallography, 5 Abbey Square, Chester CH1 2HU, England.

(v) A **data block** is the highest-level component of a CIF, containing data items or (in the case of dictionary files only) save frames. A data block is identified by a **data-block header**, which is an isolated character string (that is, bounded by white space and not forming part of a data value) beginning with the case-insensitive reserved characters `data_`. A **block code** is the variable part of a data-block header, *e.g.* the string *foo* in the header `data_foo`.

(vi) A **looped list** of data is a set of data items represented as a table or matrix of values. The data names are assembled immediately following the word `loop_`, each separated by white space, and the associated data values are then listed in strict rotation. The table of values is assembled in row-major order; that is, the first occurrence of each of the data items is assembled in sequence, then the second occurrence of each item, and so forth. In a CIF, looped lists may not be nested.

2.2.3. The syntax of a CIF

The essential syntax rules for a CIF data file are discussed alongside an example (Fig. 2.2.3.1), which is an extract from a file used to exemplify the reporting of a small-molecule crystal structure to *Acta Crystallographica Section C*. The following discussion is tutorial in nature and is intended to give an overview of the syntactic features of CIF to the general reader. The special use of save frames in dictionary files is not discussed in this summary. Software developers will find the full specification at the end of the chapter. If there are any real or apparent discrepancies between the two treatments, the full specification is to be taken as definitive.

A CIF contains only ASCII characters, organized as lines of text.

Tokens (the discrete components of the file) are separated by white space; layout is not significant. Thus, in the list of atom-site coordinates in Fig. 2.2.3.1, the hydrogen-atom entries are cosmetically aligned in columns, but the non-aligned entries for the other atoms are equally valid. Indeed, there is no requirement that each cluster of looped data values be confined to a separate row; contrast the cosmetic ordering of the atom-sites loop with the loop of symmetry-equivalent positions, where entries run on the same or following lines indiscriminately.

A comment is a token introduced by a hash character `#` and extending to the end of the line. Comments are considered to have no portable information content and may freely be discarded by a parser. However, revision 1.1 of the CIF specification introduces a *recommendation* that a CIF begin with a comment taking the form

```
#\#CIF_1.1
```

where the 1.1 is a version identifier of the reference CIF specification. This is primarily for the benefit of general file-handling software on current operating systems (*e.g.* graphical file managers that associate software applications with files of specific type), and its presence or absence does not guarantee the integrity of the file with respect to any particular revision of the CIF specification.

The first non-comment token of a CIF must be a data-block header, which is a character string that does not include white space and begins with the case-insensitive characters `data_`.

The file may be partitioned into multiple data blocks by the insertion of further data-block headers. Data-block headers are case-insensitive (that is, two headers differing only in whether corresponding letter characters are upper or lower case are considered identical). Within a single data file identical data-block headers are not permitted.

```
data_99107abs

# Chemical data
_chemical_name_systematic
; 3-Benzo[b]thien-2-yl-5,6-dihydro-1,4,2-oxathiazine
  4-oxide
;
_chemical_formula_moiety        "C11 H9 N O2 S2"
_chemical_formula_weight         251.31

# Crystal data
_symmetry_cell_setting           orthorhombic
_symmetry_space_group_name_H-M   'P 21 21 21'

loop_
   _symmetry_equiv_pos_as_xyz
 'x, y, z' 'x+1/2, -y+1/2, -z' '-x, y+1/2, -z+1/2'
 '-x+1/2, -y, z+1/2'

_cell_length_a                   7.4730(11)
_cell_length_b                   8.2860(11)
_cell_length_c                   17.527(2)
_cell_angle_alpha                90.00
_cell_angle_beta                 90.00
_cell_angle_gamma                90.00

# Atomic coordinates and displacement parameters
loop_
  _atom_site_label
  _atom_site_type_symbol
  _atom_site_fract_x
  _atom_site_fract_y
  _atom_site_fract_z
  _atom_site_U_iso_or_equiv
S4 S 0.32163(7) 0.45232(6) 0.52011(3) 0.04532(13)
S11 S 0.39642(7) 0.67998(6) 0.29598(2) 0.04215(12)
O1 O -0.00302(17) 0.67538(16) 0.47124(8) 0.0470(3)
O4 O 0.2601(2) 0.28588(16) 0.50279(10) 0.0700(5)
N2 N 0.14371(19) 0.66863(19) 0.42309(9) 0.0402(3)
C3 C 0.2776(2) 0.57587(19) 0.43683(9) 0.0332(3)
C5 C 0.1497(3) 0.5457(3) 0.57608(11) 0.0498(5)
C6 C -0.0171(3) 0.5529(2) 0.52899(12) 0.0460(4)
C12 C 0.4215(2) 0.57488(19) 0.38139(9) 0.0344(3)
C13 C 0.5830(2) 0.4995(2) 0.38737(10) 0.0386(4)
C13A C 0.6925(2) 0.5229(2) 0.32123(10) 0.0399(4)
C14 C 0.8631(3) 0.4608(3) 0.30561(13) 0.0532(5)
C15 C 0.9423(3) 0.4948(3) 0.23709(15) 0.0644(7)
C16 C 0.8563(3) 0.5917(3) 0.18349(14) 0.0667(7)
C17 C 0.6901(3) 0.6568(3) 0.19729(12) 0.0546(5)
C17A C 0.6090(3) 0.6204(2) 0.26670(10) 0.0396(4)
H5A   H    0.1284   0.4834   0.6221   0.060
H5B   H    0.1861   0.6537   0.5908   0.060
H6A   H   -0.0374   0.4490   0.5050   0.055
H6B   H   -0.1186   0.5762   0.5617   0.055
H13   H    0.6182   0.4397   0.4297   0.046
H14   H    0.9218   0.3972   0.3414   0.064
H15   H    1.0548   0.4527   0.2262   0.077
H16   H    0.9127   0.6130   0.1373   0.080
H17   H    0.6340   0.7227   0.1616   0.066
```

Fig. 2.2.3.1. Typical small-molecule CIF.

Data names are character strings that begin with an underscore character _ and do not contain white-space characters. Data names serve to index data values and are case-insensitive.

Where a data name indexes a single data value, that value follows the data name separated by white space.

Where a data name indexes a set of data values (conceptually a vector or table column), the relevant data items are preceded by the case-insensitive string `loop_` separated by white space.

The examples of Fig. 2.2.3.1 show the use of `loop_` to specify a vector or one-dimensional list of values (the symmetry-equivalent positions) and a tabular or matrix list (the atom-site positions).

Again it should be emphasized that the rows and columns of the table are identified by parsing each value and referring back to the sequence in the header of its identifying tag, and not by a conventional layout of items. It is usual for CIF writers to lay out two-dimensional data arrays as well formatted tables for ease of inspection in text editors or other viewers, but CIF readers must never rely on layout alone to identify a tabulated value.

A corollary of this is that the number of values in a looped list must be an exact integer multiple of the number of data names declared in the loop header.

Within a single data block the same data name may not be repeated. Thus if a data item may have multiple values, these items must be collected together within a looped list, the data name itself being given once only in the loop header.

A data value is a string of characters. CIF distinguishes between numerical and character data in a broad sense (see Section 2.2.5.2 below). Numerical values may not contain white space (and indeed are constrained to a limited character set and ordering, essentially encompassing a small range of ways in which numbers are characteristically represented in printed form). Because tokens are separated by white space, character data that include white-space characters must be quoted. If the data value does not extend beyond the end of a line of text, it may be quoted by matching single-quote (apostrophe, `'`) or double-quote (quotation mark, `"`) characters. If the data value does extend beyond the end of a line of text, then paired semicolon characters `;` *as the first character of a line* may be used as delimiters. The closing semicolon is the first character of a line immediately following the close of the data-value string. Fig. 2.2.3.1 shows examples of all three types of delimited character values that include white space.

Character strings that begin with certain other characters must also be quoted. These leading characters are those which introduce tokens with special roles in a STAR File (such as underscore _ at the start of a data name, hash # at the start of a comment and dollar $ identifying a save-frame reference pointer). Likewise, the STAR File reserved words `loop_`, `stop_` and `global_` must be quoted if they represent data values, as must any character string beginning with `data_` or `save_`.

Lines of text are restricted to 2048 characters in length and data names are restricted to 75 characters in length. These are increases over the original values of 80 and 32 characters, respectively.

2.2.4. Portability and archival issues

The CIF format is designed to be independent of operating system (OS) and programming language. Nevertheless, variations in the way that each OS specifies and handles character sets mean that care must be taken to ensure that CIF software is portable across different computer platforms. There are also constraints on the application of these specifications in order to maintain compatibility between archival systems. These issues are discussed briefly here. More details are given in the formal CIF specification (see Section 2.2.7). In general, compatibility and portability considerations for different OSs are of little importance to users of CIFs, but they need to be well understood by software developers.

2.2.4.1. Character set

The characters permitted in a CIF are in effect the printable characters in the ASCII character set. However, a CIF may also be constructed and manipulated using alternative single-character byte mappings such as EBCDIC, and multi-byte or wide character encodings such as Unicode, provided there is a direct mapping to the permitted ASCII characters. Accented characters, characters in non-Latin alphabets and mathematical or special typographic symbols may not appear as single characters in a CIF, even if a host OS permits such representations.

White space (used to separate CIF tokens and within comments or quoted character-string values) is most portably represented by the printable space character (decimal value 32 in the ASCII character set). In an ASCII environment, white space may also be indicated by the control characters denoted HT (horizontal tab, ASCII decimal 9), LF (line feed, ASCII decimal 10) and CR (carriage return, ASCII decimal 13). To ease problems of translation between character encodings, the characters VT (vertical tab, ASCII decimal 11) and FF (form feed, ASCII decimal 12) are explicitly excluded from the CIF character set; this is a restriction that is not in the general STAR File specification (Chapter 2.1).

2.2.4.2. Line terminators

Given that the STAR File is built on the premise of a line-oriented text file, it is difficult in practice to provide a complete and portable description of how to identify the start or end of a line of text. The difficulty arises for two reasons.

First, some OSs or programming languages are record-oriented; that is, the OS is able to keep track of a region of memory associated with a specific record. It is usually appropriate to associate each such record with a line of text, but the 'boundaries' between records are managed by low-level OS utilities and are not amenable to a character-oriented discussion. Such systems, where records of fixed length are maintained, may also give rise to ambiguities in the interpretation of padding to the record boundary following a last printable character – is such padding to be discarded or treated as white space? It is for this reason that the elision of trailing white space in a line is permitted (but not encouraged) in the full CIF syntax specification [Section 2.2.7.1.4(17)].

The second complication arises because current popular OSs support several different character-based line terminators. Historically, applications developed under a specific OS have made general use of system libraries to handle text files, so that the conventions built into the system libraries have in effect become standard representations of line terminators for all applications built on that OS. For a long time this created no great problem, since files were transferred between OSs through applications software that could be tuned to perform the necessary line-terminator translations in transit. The best-known such application is undoubtedly the 'text' or 'ascii' transmission mode of the typical ftp (file transfer protocol) client.

Increasingly, however, a common network-mounted file system may be shared between applications running under different OSs and the same file may present itself as 'valid' to one user but not to another because of differences in what the applications consider a line terminator.

The problem of handling OS-dependent line termination is by no means unique to STAR File or CIF applications; any application that manipulates line-oriented text files must accommodate this difficulty. The specification notes the practice of designing applications that treat equivalently as a line terminator the characters LF (line feed or newline), CR (carriage return) or the combination CR followed by LF, since these are the dominant conventions under the prevailing Unix, MacOS and DOS/Windows OSs of the present day. While this will be a sensible design decision for many CIF-reading applications, software authors must be aware that the CIF specification aims for portability and archivability through a more general understanding of what constitutes a line of text.

2.2.4.3. Line lengths

The STAR File does not restrict the lengths of text lines. The original CIF specification introduced an 80-character limit to facilitate programming in Fortran and transmission of CIFs by email. Contemporary practices have enabled the line-length limit in the version 1.1 specification to be extended to 2048 characters. While the new limit in effect mandates the use of a 2048-character input buffer in compliant CIF-reading software, there is no obligation on CIF writers to generate output lines of this length.

2.2.4.4. Lengths of data names and block codes

CIF imposes another length restriction that is not integral to the STAR File syntax: data names, data-block codes and frame codes may not exceed 75 characters in length. This is an increase over the 32-character limit of the original specification, although this extension had already been approved by COMCIFS to coincide with the release of version 2 of the core dictionary (see Chapter 3.2).

There is no fundamental technical reason underlying this restriction; it permits assignment of a fixed-length buffer for recording such data names within a software application, but perhaps more significantly, it encourages a measure of conciseness in the creation of data names based on hierarchical component terms.

2.2.4.5. Case sensitivity

Following the general STAR File approach, the special tokens `loop_`, `save_`, the reserved word `global_`, data-block codes and save-frame codes, and data names are all case-insensitive. The case of any characters within data values must, however, be respected.

2.2.5. Common semantic features

As mentioned in Section 2.2.1, the STAR File structure allows retrieval of indexed data values without prior knowledge of the location of a data item within the file. Consequently there is no significance to the order of data items within a data block. However, molecular-structure applications will typically need to do more than retrieve an arbitrary string value. There is a need to identify the nature of individual data items (achieved portably through the definitions of standard data names in dictionary files), but also to be able to process the extracted data according to whether it is numerical or textual in nature, and possibly also to parse and extract more granular information from the entire data field that has been retrieved.

Different CIF applications have a measure of freedom to define many of the details of the content of data fields and the ways in which they may be processed – in effect, to define their semantic content. However, there are a number of conventions that are common to all CIF applications and this should be recognized in software applicable to a range of dictionaries.

These are discussed in some detail in the formal specification document in Section 2.2.7.4; in this section some introductory comments and additional explanations are given.

2.2.5.1. Data-name semantics

It is a fundamental principle of the STAR File approach that a data name is simply an arbitrary string acting as an index to a required value or set of values. It is equally legitimate to store the value of a crystal cell volume either as

```
_bdouiGFG78=z  1085.3(3)
```

or as

```
_cell_volume   1085.3(3)
```

provided that the users of the file have some way of discovering that the cell volume is indeed indexed by the tag `_bdouiGFG78=z` or `_cell_volume`, as appropriate.

However, it is conventional in CIF applications to define (in public dictionary files) data names that imply by their construction the meaning of the data that they index. Chapter 3.1 discusses the principles that are recommended for constructing data names and defining them in public dictionaries, and for utilizing private data names that will not conflict with those in the public domain.

Careful construction of data names according to the principles of Chapter 3.1 results in a text file that is intelligible to a scientist browsing it in a text editor without access to the associated dictionary definition files. In many ways this is useful; it allows the CIF to be viewed and understood without specialized software tools, and it safeguards some understanding of the content if the associated dictionaries cannot be found. On the other hand, there is a danger that well intentioned users may gratuitously invent data names that are similar to those in public use. It is therefore important for determining the correct semantic content of the values tagged by individual data names to make maximum possible disciplined use of the registry of public dictionaries, the registry of private data-name prefixes, and the facilities for constructing and disseminating private dictionaries discussed in Chapter 3.1.

2.2.5.2. Data typing

In the STAR File grammar, all data values are represented as character strings. CIF applications may define data types, and in the macromolecular (mmCIF) dictionary (see Chapter 3.6) a range of types has been assigned corresponding to certain contemporary computer data-storage practices (*e.g.* single characters, case-insensitive single characters, integers, floating-point numbers and even dates). This dynamic type assignment is supported by the relational dictionary definition language (DDL2; see Chapter 2.6) used for the mmCIF dictionary and is not available for all CIF applications.

However, a more restricted set of four primary or base data types is common to all CIF applications.

The type **numb** encompasses all data values that are interpretable as numeric values. It includes without distinction integers and non-integer reals, and the values may be expressed if desired in scientific notation. At this revision of the specification it does not include imaginary numbers. All numeric representations are understood to be in the number base 10.

It is, however, a complex type in that the standard uncertainty in a measured physical value may be carried along as part of the value. This is denoted by a trailing integer in parentheses, representing the integer multiple of the uncertainty in the last place of decimals in the numeric representation. That is, a value of ‘1085.3(3)’ corresponds to a measurement of 1085.3 with a standard uncertainty of 0.3. Likewise, the value 34.5(12) indicates a standard uncertainty of 1.2 in the measured value.

Care should be taken in the placement of the parentheses when a number is expressed in scientific notation. The second example above may also be presented as 3.45E1(12); that is, the standard uncertainty is applied to the mantissa and not the exponent of the value.

Note that existing DDL2 applications itemize standard uncertainties as separate data items. Nevertheless, since the DDL2 dictionary includes the attribute `_item_type_conditions.code` with an allowed value of ‘esd’, future conformant DDL2 parsers might be expected to handle the parenthesized standard uncertainty representation.

The preferred behaviour of a CIF application is to determine the type of a data value by looking up the corresponding dictionary definition. However, some CIF-reading software may not be designed with the ability to parse dictionaries; and indeed any CIF reader may encounter data names that are not defined in a public or accompanying dictionary. It is therefore appropriate to adopt a strategy of interpreting as a number any data value that looks like one, *i.e.* adopts any of the permitted ways to represent a numeric value. Therefore, in the absence of a specific counter-indication (from a dictionary definition), the data value in the following example may be taken as the numeric (integer) value 1:

```
_unknown_data_name    1
```

On the other hand, if `_unknown_data_name` were explicitly defined in a dictionary with a data type of 'char', then the value should be stored as the literal character `1`.

This is a subtle point, perhaps of interest only to software authors. Nevertheless, the consistent behaviour of CIF applications will depend on correct implementation of this behaviour.

The data type **char** covers single characters or extended character strings. Since CIF tokens are separated by white space, any character string that includes white-space characters (including line-terminating characters) must be delimited by one or other of a set of special characters used for this purpose. The detailed rules for quoting such strings are given in Section 2.2.7.1.4 and comprise the standard CIF syntax rules for this case. No semantic distinction is made in general between short character strings and text strings that extend over several lines, described in the specification document as 'text fields', although again particular CIF applications may choose to impose distinctions. Note that numbers within a quoted string or a text block (bounded by semicolons in column 1) are not interpreted as type 'numb' but as type 'char'.

The data type **uchar** was introduced explicitly at revision 1.1 of the CIF specification, and is intended to formalize the description and automated handling of certain strings in CIFs that are case-insensitive (such as data names and data-block headers).

The data type **null** is a special type that has two uses. It is applied to items for which no definite value may be stored in computer memory. As such it is a formal device for allowing the introduction of data names into dictionary files that do not represent data values permissible within a data file instance. The usual example is that of the special data names introduced in DDL1 dictionaries (such as the core dictionary) to discuss categories.

The more important use of the null data type is its application to the meta characters '`?`' (query) and '`.`' (full point) that may occur as values associated with any data name and therefore have no specific type. (Arguably, for this case 'any' might be a better type descriptor than 'null'.)

The substitution of the query character '`?`' in place of a data value is an explicit signal that an expected value is missing from a CIF. This 'missing-value signal' may be used instead of omitting an item (*i.e.* its tag and value) entirely from the file, and serves as a reminder that the item would normally be present.

The substitution of the full-point character '`.`' in place of a CIF data value serves two similar, but not identical, purposes. If it is used in looped lists of data it is normally a signal that a value in a particular packet (*i.e.* a value in the row of the table) is 'inapplicable' or 'inappropriate'. In some CIF applications involving access to a data dictionary it is used to signal that the default value of the item is defined in its definition in the dictionary. Consequently, the interpretation of this signal is an application-specific matter and its use must be determined according to the application. For example, in a CIF submitted for publication in *Acta Crystallographica* the presence of a '`.`' value for the item `_geom_bond_site_symmetry_1` is predetermined as the default value 1_555 (as per the dictionary definition). Note that, in this instance, it is also equivalent to 'no additional symmetry' or 'inapplicable'.

2.2.5.3. Extended data typing: content type and encoding

The initial implementation of CIF assumed that most character strings would represent identifiers or terse descriptions or comments, and that the correct behaviour of the majority of CIF applications would be simply to store these in computer memory or retrieve them verbatim. Only a few data values were foreseen as having extended content that might need special handling. For example, the complete text of a manuscript was envisaged as being included in the field `_publ_manuscript_processed`. The handling of this field (its extraction and typesetting) would be left to unspecified external agents, although some clue as to the provenance of the contents of that field (and thus their appropriate handling) would be given by `_publ_manuscript_creation`.

However, the evolution of CIF applications has required that some element of typographic markup be permitted in a growing number of data values, and future applications may be envisaged in which graphical images, virtual-reality models, spreadsheet tables or other complex objects are embedded as the values of specific data items. Since it will not be possible to write general-purpose CIF applications capable of handling all such embedded content, techniques will need to be developed for transferring each such field to a specialized but separate content handler. In the meantime, the rather *ad hoc* conventions for introducing typographic markup available at present are described in Sections 2.2.7.4.13–17. It is hoped that in the future different types of such markup may be permitted so long as the data values affected can be tagged with an indication of their content type that allows the appropriate content handlers to be invoked.

It has also been necessary to allow native binary objects to be incorporated as CIF data values. This was done to support the storage of the large arrays of image data obtained from area detectors. Since the CIF character set is based on printable ASCII characters only, encodings including compression have been developed to permit interconversion between ASCII and binary representations of such data (see Chapter 2.3).

Nowadays, arbitrary embedded objects may be transported in web pages *via* the http protocol (Fielding *et al.*, 1999) or as attachments to email messages structured according to the MIME protocols (*e.g.* Freed & Borenstein, 1996). Identification of encoding techniques and hooks to invoke suitable handlers are carried in the relevant Content-Type and Content-Encoding http or MIME headers. It is suggested that this may form the basis of suitable tagging of content types and encoding for future CIF development.

A candidate for a CIF-specific encoding protocol is the special convention introduced with CIF version 1.1 to interconvert long lines of text between the new and old length limits (Section 2.2.7.4.11). This is an encoding in the sense that it is a device designed to retain any semantic content implicit in textual layout, while conforming to slightly different rules of syntax. It is designed to enable CIFs written to the longer line-length specification to be transformed so that they can still be handled by older software. Since the object of the exercise is to manage legacy applications, it is likely that the interconversion will be done through external applications, or filters, designed specifically for the purpose. Such a conversion filter is conceptually the same as a filter to convert a binary file into an ASCII base-64 encoding, for example.

2.2.6. CIF metadata and dictionary compliance

The development of several CIF dictionaries and fields of application has rapidly progressed beyond the specific purpose of describing a small-molecule or inorganic crystal structure for which CIF was devised. With these, a variety of application-specific metadata approaches have evolved to characterize the role of a particular CIF within a family of possible applications. These approaches use data definitions in dictionaries in which enumerated codes identify the file relationships. The mmCIF dictionary (see Chapter 3.6) allows informal identification of 'external reference files' which act as libraries of standard molecular geometry. The pdCIF dictionary (see Chapter 3.3) specifies identifiers that may be included within data blocks of external files containing calibration results. It is the responsibility of the file users to manage a lookup table or database between the referenced identifiers and the location of the files to which they pertain.

Two categories of data items currently exist in the core dictionary to allow a file to indicate its relationship to CIF dictionaries and other data files. (Equivalent categories are also present in the mmCIF dictionary.) AUDIT_CONFORM is a category of data names identifying the dictionaries that hold definitions of the data names in the current CIF. Particularly where the referenced dictionaries include any of the various public dictionaries described in Part 3 of this volume, this serves to establish the discipline within the broad fields of crystallography, structural biology and structural chemistry to which the data are most relevant.

The category AUDIT_LINK allows an informal textual description of the relationship between the data blocks within the current file. It is 'informal' in the sense that the relevant data items are free-text in nature. It would surely be useful to have a catalogue of more specific designations to allow automated software to track such relationships as the separate reference and modulated structures in an incommensurate compound, or the multiple trial refinements of a protein structure. The challenge is to determine and classify such standard relationships between data blocks.

In the future it is hoped that a common approach to metadata will be developed to enable all CIF instantiations to be uniquely identified and interrelated. Development of standard descriptions of the relationships between structural entities of this sort (reference geometries, calibration results, partial refinements, modulated superposed structures *etc.*) will be an important stage in the formalization of complete CIF metadata, and will become an important step towards categorization of data entities needed for interoperability between different file formats and across a wide range of scientific disciplines.

2.2.7. Formal specification of the Crystallographic Information File

Version 1.1 specification

By S. R. Hall, N. Spadaccini, I. D. Brown,
H. J. Bernstein, J. D. Westbrook and B. McMahon

This section presents the documents *File syntax* (Sections 2.2.7.1–3) and *Common semantic features* (Section 2.2.7.4) that together comprise the formal CIF specification as approved by COMCIFS.

2.2.7.1. Syntax

2.2.7.1.1. *Introduction*

(1) This document describes the full syntax of the Crystallographic Information File (CIF).

2.2.7.1.2. *Definition of terms*

(2) The following terms are used in the CIF specification documents with the specific meanings indicated here.

(2.1) A **CIF** is a file conforming to the specification herein stated, containing either information on a crystallographic experiment or its results (or similar scientific content), or descriptions of the data identifiers in such a file.

(2.2) A **data file** is understood to convey information relating to a crystallographic experiment.

(2.3) A **dictionary file** is understood to contain information about the data items in one or more data files as identified by their data names.

(2.4) A **data name** is a case-insensitive identifier (a string of characters beginning with an underscore character) of the content of an associated data value.

(2.5) A **data value** is a string of characters representing a particular item of information. It may represent a single numerical value; a letter, word or phrase; extended discursive text; or in principle any coherent unit of data such as an image, audio clip or virtual-reality object.

(2.6) A **data item** is a specific piece of information defined by a data name and an associated data value.

(2.7) A **tag** is understood in this document to be a synonym for data name.

(2.8) A **data block** is the highest-level component of a CIF, containing data items or save frames. A data block is identified by a **data-block header**, which is an isolated character string (that is, bounded by white space and not forming part of a data value) beginning with the case-insensitive reserved characters `data_`.

(2.9) A **block code** is the variable part of a data-block header, *e.g.* the string `foo` in the header `data_foo`.

(2.10) A **save frame** is a partitioned collection of data items within a data block, started by a **save-frame header**, which is an isolated character string beginning with the case-insensitive reserved characters `save_`, and terminated with an isolated character string containing only the case-insensitive reserved characters `save_`.

(2.11) A **frame code** is the variable part of a save-frame header, *e.g.* the string `foo` in the header `save_foo`.

2.2.7.1.3. *File syntax*

(3) The syntax of CIF is a proper subset of the syntax of STAR Files as described by Hall (1991) and Hall & Spadaccini (1994). The general structure is described below in Section 2.2.7.1.4 and a number of subsections list specific restrictions to the STAR syntax that are in force within CIF. A formal language grammar using computer-science notation is included as Section 2.2.7.2.

2.2.7.1.4. *General features*

(4) A CIF consists of **data names** (tags) and associated values organized into **data blocks**. A data block may contain **data items** (associated data names and data values) and/or it may contain **save frames**.

(5) **Save frames** may only be used in dictionary files.

Implementation note: At a purely syntactic level there is no way to distinguish between dictionary and data files. (It is also to be noted that not all dictionary files contain save frames.) A fully validating parser must therefore be able to detect the start and termination of save frames, the uniqueness of the frame code within a data block and the uniqueness of data names within a frame code. It is, however, legitimate for an application-based parser designed to handle only the contents of data files to consider the presence of a save frame as an error.

(6) A **data block** begins with the reserved case-insensitive string `data_` followed immediately by the name of the data block, forming a **data-block header**. A **save frame** has a similar structure to a data block, but may not itself contain further save frames. A save frame begins with the reserved case-insensitive string `save_` followed immediately by the name of the save frame, forming a **save-frame header**. Unlike a data block, a save frame also has a marker for the end of the frame in the form of a repetition of the reserved case-insensitive word `save_` , this time without the name of the frame. Save frames may not nest. Within a single CIF, no two data blocks may have the same name; within a single data block no two save frames may have the same name, although a save frame may have the same name as a data block in the same CIF.

(7) A given **data name** (tag) [see (2.4) and (2.7)] may appear no more than once in a given data block or save frame. A tag may be followed by a single value, or a list of one or more tags may be marked by the preceding reserved case-insensitive word `loop_` as the headings of the columns of a table of values. White space is used to separate a data-block or save-frame header from the contents of the data block or save frame, and to separate tags, values and the reserved word `loop_`. Data items (tags along with their associated values) that are not presented in a table of values may be relocated along with their values within the same data block or save frame without changing the meaning of the data block or save frame. Complete tables of values (the table column headings along with all columns of data) may be relocated within the same data block or save frame without changing the meaning of the data block or save frame. Within a table of values, each tag may be relocated along with its associated column of values within the same table of values without changing the meaning of the table of values. In general, each row of a table of values may also be relocated within the same table of values without changing the meaning of the table of values. Combining tables of values or breaking up tables of values would change the meanings, and is likely to violate the rules for constructing such tables of values.

(8) The case-insensitive word `global_` , used in STAR Files to introduce a group of data values with a scope extending to the end of the file, is an additional reserved word in CIF (that is, it may not be used as the unquoted value of any data item).

(9) If a **data value** (2.5) contains white space or *begins* with a character string reserved for a special purpose, it *must* be delimited by one of several sets of special character strings (the choice of which is constrained if the data value contains characters interpretable as marking a new line of text according to the discussion in the following paragraphs). Such a data value will be indicated by the term *non-simple data value*.

(10) A *simple* data value (*i.e.* one which does not contain white space or begin with a special character string) may optionally be delimited by any of the same set of delimiting character strings, *except* for data values that are to be interpreted as numbers.

(11) The special character strings in this context are listed in the following table. The term 'non-simple data values' in this table refers to data values beginning with these special character strings.

Character or string	Role
`_`	identifies data name
`#`	identifies comment
`$`	identifies save-frame pointer
`'`	delimits non-simple data values
`"`	delimits non-simple data values
`[`	reserved opening delimiter for non-simple data values [see (19)]
`]`	reserved closing delimiter for non-simple data values [see (19)]
`;` (at the beginning of a line of text)	delimits non-simple data values
`data_`	identifies data-block header
`save_`	identifies save-frame header or terminator

In addition, the following case-insensitive *reserved words* may not occur as unquoted data values.

Reserved word	Role
`loop_`	identifies looped list of data
`stop_`	reserved STAR word terminating nested loops or loop headers
`global_`	reserved as a STAR global-block header

(12) The complete syntactic description of a numeric data value is included in Section 2.2.7.3(57) under the production (*i.e.* rule for constructing a part of the language) `<Numeric>`.

(13) *Comment:* The base CIF specification distinguishes between character and numeric values [see Section 2.2.7.4(15)]. Particular CIF applications may make more finely grained distinctions within these types. The paragraphs immediately above have the corollary that a data value such as `12` that appears within a CIF may be quoted (*e.g.* `'12'`) *if and only if* it is to be interpreted and stored in computer memory as a character string and not a numeric value. For example `'12'` might legitimately appear as a label for an atomic site, where another alphabetic or alphanumeric string such as `'C12'` is also acceptable; but it may *not* legitimately be used to represent an integer quantity twelve.

(14) Matching single- or double-quote characters (`'` or `"`) may be used to bound a string representing a non-simple data value *provided* the string does not extend over more than one line.

(15) *Comment:* Because data values are invariably separated from other tokens in the file by white space, such a quote-delimited character string may contain instances of the character used to delimit the string *provided* they are not followed by white space. For example, the data item

```
_example  'a dog's life'
```

is legal; the data value is `a dog's life`.

(16) *Comment:* Note that constructs such as

```
'an embedded \' quote'
```

do *not* behave as in the case of many current programming languages, *i.e.* the backslash character in this context does not escape the special meaning of the delimiter character. A backslash preceding the apostrophe or double-quote characters does, however, have special meaning in the context of accented characters (Section 2.2.7.4.15) provided there is no white space immediately following the apostrophe or double-quote character.

(17) The special sequence of end of line followed immediately by a semicolon in column one (denoted '`<eol>;`') may also be used as a delimiter at the beginning and end of a character string comprising a data value. The complete bounded string is called a **text field** and may be used to convey multi-line values. The end of line associated with the closing semicolon does *not* form part of the data value. Within a multi-line text field, leading white space within text lines must be retained as part of the data value; trailing white space on a line may however be elided.

(18) *Comment:* A text field delimited by the `<eol>;` digraph *may not* include a semicolon at the start of a line of text as part of its value.

(19) Matching square-bracket characters, '[' and ']', are *reserved* for possible future introduction as delimiters of multi-line data values. At this revision of the CIF specification, a data value may not begin with an unquoted left square-bracket character '['. (While not strictly necessary, the right square-bracket character ']' is restricted in the same way in recognition of its reserved use as a closing delimiter.)

(20) *Comment:* For example, the data value `foo` may be expressed equivalently as an unquoted string `foo`, as a quoted string `'foo'` or as a text field

```
;foo
;
```

By contrast, the value of the text field

```
; foo
  bar
;
```

is

```
 foo<eol>  bar
```

(where `<eol>` represents an end of line); the embedded space characters are significant.

(21) A comment in a CIF begins with an unquoted character '`#`' and extends to the end of the current line.

2.2.7.1.5. *Character set*

(22) Characters within a CIF are restricted to certain printable or white-space characters. Specifically, these are the ones located in the ASCII character set at decimal positions 09 (HT or horizontal tab), 10 (LF or line feed), 13 (CR or carriage return) and the letters, numerals and punctuation marks at positions 32–126.

Comment: The ASCII characters at decimal positions 11 (VT or vertical tab) and 12 (FF or form feed), often included in library implementations as white-space characters, are explicitly excluded from the CIF character set at this revision.

(23) *Comment:* The reference to the ASCII character set is specifically to identify characters in an established and widely available standard. It is understood that CIFs may be constructed and maintained on computer platforms that implement other character-set encodings. However, for maximum portability only the characters identified in the section above may be used. Other printable characters, even if available in an accessible character set such as Unicode, must be indicated by some encoding mechanism using only the permitted characters. At this revision, only the encoding convention detailed in Section 2.2.7.4(30)–(37) is recognized for this purpose.

2.2.7.1.6. *White space*

(24) Any of the white-space characters listed in paragraph (22) (*i.e.* HT, LF, CR) and the visible space character SP (position number 32 in the ASCII encoding) may be used interchangeably to separate tokens, with the exception that the semicolon characters delimiting multi-line text fields must be preceded by the white-space character or characters understood as indicating an end of line (see next paragraph).

2.2.7.1.7. *End-of-line conventions*

(25) The way in which a line is terminated is operating-system dependent. The STAR File specification does not address different operating-system conventions for encoding the end of a line of text in a text file. For a file generated and read in the same machine environment, this is rarely a problem, but increasingly applications on a network host may access files on different hosts through protocols designed to present a unified view of a file system. In practice, for current common operating systems many applications may regard the ASCII characters LF or CR or the sequence CR LF as signalling an end of line, inasmuch as these represent the end-of-line conventions supported under the common operating systems Unix, MacOS or DOS/Windows. On platforms with record-oriented operating systems, applications must understand and implement the appropriate end-of-line convention. Care must be taken when transferring such files to other operating systems to insert the appropriate end-of-line characters for the target operating system. A more complete discussion is given in (42) below.

2.2.7.1.8. *Case sensitivity*

(26) Data names, block and frame codes, and reserved words are case-insensitive. The case of any characters within data values must be respected.

2.2.7.1.9. *Implementation restrictions*

(27) Certain allowed features of STAR File syntax have been expressly excluded or restricted from the CIF implementation.

2.2.7.1.9.1. *Maximum line length and character set*

(28) Lines of text may not exceed 2048 characters in length. This count excludes the character or characters used by the operating system to mark the line termination.

The ASCII characters decimal 11 (VT) and 12 (FF) are excluded from the allowed character set [see paragraph (22)].

2.2.7.1.9.2. *Maximum data-name, block-code and frame-code lengths*

(29) Data names may not exceed 75 characters in length.

(30) Data-block codes and save-frame codes may not exceed 75 characters in length (and therefore data-block headers and save-frame headers may not exceed 80 characters in length).

2.2.7.1.9.3. *Single-level loop constructs*

(31) Only a single level of looping is permitted.

2.2.7.1.9.4. *Non-expansion of save-frame references*

(32) Save frames are permitted in CIFs, but expressly for the purpose of encapsulating data-name definitions within data dictionaries. No reference to these save frames is envisaged, and the save-frame reference code permitted in STAR is not used. This means that unquoted character strings commencing with the `$` character may not be interpreted as save-frame codes in CIF. Use of such unquoted character strings is *reserved* to guard against subsequent relaxation of this constraint.

2.2.7.1.9.5. *Exclusion of global_ blocks*

(33) In the full STAR specification, blocks of data headed by the special case-insensitive word `global_` are permitted before normal data blocks. They contain data names and associated values which are inherited in subsequent data blocks; the scope of a value extends from its point of declaration in a global block to the end of the file. Because rearrangements of the order of data blocks and concatenation of data blocks from different files are commonplace operations in many CIF applications, and because of the difficulty in properly tracking and implementing values implied by global blocks, use of the `global_` feature of STAR is expressly *forbidden* at this revision. To guard against its future introduction, the special case-insensitive word `global_` remains *reserved* in CIF.

Table 2.2.7.1. *A formal grammar for CIF*

(*a*) Basic structure of a CIF.

Syntactic unit	Syntax	Case sensitive?
<CIF>	<Comments>? <WhiteSpace>? {<DataBlock> {<WhiteSpace> <DataBlock> }* {<WhiteSpace> }? }?	yes
<DataBlock>	<DataBlockHeading> {<WhiteSpace> { <DataItems> | <SaveFrame>} }*	yes
<DataBlockHeading>	<DATA_> {<NonBlankChar> }+	no
<SaveFrame>	<SaveFrameHeading> {<WhiteSpace> <DataItems> }+ <WhiteSpace> <SAVE_>	yes
<SaveFrameHeading>	<SAVE_> {<NonBlankChar> }+	no
<DataItems>	<Tag> <WhiteSpace> <Value> | <LoopHeader> <LoopBody>	yes
<LoopHeader>	<LOOP_> {<WhiteSpace> <Tag>}+	no
<LoopBody>	<Value> {<WhiteSpace> <Value> }*	yes

(*b*) Reserved words.

Syntactic unit	Syntax	Case sensitive?
<DATA_>	{’D’|’d’} {’A’|’a’} {’T’|’t’} {’A’|’a’} ’_’	no
<LOOP_>	{’L’|’l’} {’O’|’o’} {’O’|’o’} {’P’|’p’} ’_’	no
<GLOBAL_>	{’G’|’g’} {’L’|’l’} {’O’|’o’} {’B’|’b’} {’A’|’a’} {’L’|’l’} ’_’	no
<SAVE_>	{’S’|’s’} {’A’|’a’} {’V’|’v’} {’E’|’e’} ’_’	no
<STOP_>	{’S’|’s’} {’T’|’t’} {’O’|’o’} {’P’|’p’} ’_’	no

(*c*) Tags and values.

Syntactic unit	Syntax	Case sensitive?
<Tag>	’_’{<NonBlankChar>}+	no
<Value>	{’.’ | ’?’ | <Numeric> | <CharString> | <TextField> }	yes

(*d*) Numeric values.

Syntactic unit	Syntax	Case sensitive?
<Numeric>	{<Number> | <Number> ’(’ <UnsignedInteger> ’)’ }	no
<Number>	{<Integer> | <Float> }	no
<Integer>	{{ ’+’ | ’-’ }? <UnsignedInteger>	no
<Float>	{ <Integer><Exponent> | { {’+’|’-’} ? { {<Digit>} * ’.’ <UnsignedInteger> } | { <Digit>} + ’.’ } } {<Exponent>} ? } }	no
<Exponent>	{ {’e’ | ’E’ } | {’e’ | ’E’ } { ’+’ | ’-’ } } <UnsignedInteger>	no
<UnsignedInteger>	{<Digit> }+	no
<Digit>	{ ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’ }	no

(*e*) Character strings and text fields.

Syntactic unit	Syntax	Case sensitive?
<CharString>	<UnquotedString> | <SingleQuotedString> | <DoubleQuotedString>	yes
<eol><UnquotedString>	<eol><OrdinaryChar> {<NonBlankChar>}*	yes
<noteol><UnquotedString>	<noteol>{<OrdinaryChar>|’;’} {<NonBlankChar>}*	yes
<SingleQuotedString><WhiteSpace>	<single_quote>{<AnyPrintChar>}* <single_quote> <WhiteSpace>	yes
<DoubleQuotedString><WhiteSpace>	<double_quote> {<AnyPrintChar>}* <double_quote> <WhiteSpace>	yes
<TextField>	{<SemiColonTextField> }	yes
<eol><SemiColonTextField>	<eol>’;’ { {<AnyPrintChar>}* <eol> {{<TextLeadChar> {<AnyPrintChar>}*}? <eol>}* } ’;’	yes

Table 2.2.7.1. *(cont.)*

(*f*) White space and comments.

Syntactic unit	Syntax	Case sensitive?
<WhiteSpace>	{<SP>|<HT>|<eol>|<TokenizedComments>}+	yes
<Comments>	{ ’#’ {<AnyPrintChar>}* <eol>}+	yes
<TokenizedComments>	{<SP>|<HT>|<eol>|}+ <Comments>	yes

(*g*) Character sets.

Syntactic unit	Syntax	Case sensitive?
<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’|’{’|’|’|’}’|’~’ }	yes
<NonBlankChar>	<OrdinaryChar>|<double_quote>|’#’|’$’|<single_quote>|’_’ |’;’|’[’|’]’	yes
<TextLeadChar>	<OrdinaryChar>|<double_quote>|’#’|’$’|<single_quote>|’_’|<SP>|<HT>|’[’|’]’	yes
<AnyPrintChar>	<OrdinaryChar>|<double_quote>|’#’|’$’|<single_quote>|’_’|<SP>|<HT>|’;’|’[’|’]’	yes

2.2.7.1.10. *Version identification*

(34) As an archival file format, the CIF specification is expected to change infrequently. Revised specifications will be issued to accompany each substantial modification. A CIF may be considered compliant against the most recent version for which in practice it satisfies all syntactic and content rules as detailed in the formal specification document. However, to signal the version against which compliance was claimed at the time of creation, or to signal the file type and version to applications (such as operating-system utilities), it is *recommended* that a CIF begin with a structured comment that identifies the version of CIF used. For CIFs compliant with the current specification, the first 11 bytes of the file should be the string

```
#\#CIF_1.1
```

immediately followed by one of the white-space characters permitted in paragraph (22).

2.2.7.2. A formal grammar for CIF

2.2.7.2.1. *Summary*

(35) The rows of Table 2.2.7.1 are called ‘productions’. Productions are rules for constructing sentences in a language. They are written in terms of ‘terminal symbols’ and ‘non-terminal symbols’. ‘Terminal symbols’ are what actually appear in a language. For example, `’poodle’` might be given as a string of terminal symbols in some language discussing dogs. Non-terminal symbols are the higher-level constructs of the language, *e.g.* sentences, clauses, *etc.* For example `<DOG>` might be given as a non-terminal symbol in some language discussing dogs. Productions may be used to infer rules for parsing the language. For example,

```
<DOG> ::= { ’poodle’|’terrier’|’bulldog’|’greyhound’ }
```

might be given as a rule telling us what names of types of dogs we are allowed to write in this language. In this table, terminal symbols (*i.e.* terminal character strings) are enclosed in single quotes. To avoid confusion, the terminal symbol consisting of a single quote (*i.e.* an apostrophe) is indicated by `<single_quote>` and the terminal symbol consisting of a double quote is indicated by `<double_quote>`. The printable space character is indicated by `<SP>`, the horizontal tab character by `<HT>` and the end of a line by `<eol>`. To allow for the occurrence of a semicolon as the initial character of an unquoted character string, provided it is not the first character in a line of text, the special symbol `<noteol>` is used below to indicate any character that is not interpretable as a line terminator. The cases of context sensitivity involving the beginning of text fields and the ends of quoted strings are discussed below, but they are most commonly resolved in a lexical scan.

(36) Productions can be used to produce documents, or equivalently to check a document to see if it is valid in this grammar. The angle brackets delimit names for the syntactic units (the ‘non-terminal symbols’) being defined. The curly braces enclose alternatives separated by vertical bars and/or followed by a plus sign for ‘one or more’, an asterisk for ‘zero or more’ or a question mark for ‘zero or one’.

(37) In most cases, each production has a single non-terminal symbol in the syntactic unit being defined. However, in some cases, both the syntactic unit and the syntax begin or end with some common symbol. This indicates that a specific context is required in order for the rule to be applied. This is done because the initial semicolon of a semicolon-delimited text field only has meaning at the beginning of a line, and quoted strings may contain their initial quoting character provided the embedded quoting character is not immediately followed by white space. This ‘context-sensitive’ notation is unusual in defining computer languages (although very common in the full specifications of many computer and non-computer languages). This context-sensitive notation greatly simplifies the definitions and is simple to implement. The formal definitions are elaborated below.

(38) In the present revision, the production for `<TextField>` is a trivial equivalence to `<SemiColonTextField>`. The redundancy is retained to permit possible future extensions to text fields, in particular the possible introduction of a bracket-delimited text value.

2.2.7.2.2. *Explanation of the formal syntax*

Comment: Readers not familiar with the conventions used in describing language grammars may wish to consult various lecture notes on the subject available on the web, *e.g.* Bernstein (2002).

(39) In creating a parser for CIF, the normal process is to first perform a ‘lexical scan’ to identify ‘tokens’ in the CIF. A ‘token’ is a grammatical unit, such as a special character, or a tag or a value, or some major grammatical subunit. In the course of a lexical scan, the input stream is reduced to manageable pieces, so that

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

cell, but it should be emphasized from a computer-programming viewpoint that this is coincidental; the attributes that constrain the value of this data item (and its relationship to others such as `_atom_site_fract_y` and `_atom_site_fract_z`) must be obtained from the dictionary and not otherwise inferred.

(8) *Comment:* In practice data names described in a DDL2 dictionary are constructed with a period character separating their specific function from the name of the category to which they have been assigned. In the absence of a dictionary file, this convention permits the inference that the data item with name `_atom_site.fract_x` will appear in the same looped list as other items with names beginning `_atom_site.`, and that all such items belong to the same category.

2.2.7.4.5. *Name space*

(9) The intention of the maintainers of public CIF dictionaries is to formulate a single authoritative set of data names for each CIF dialect (*i.e.* DDL1 and DDL2), thus facilitating the reliable archive and interchange of crystallographic data. However, it is also permissible for users to introduce local data names into a CIF. Two mechanisms exist to reduce the danger of collision of data names that are not incorporated into public dictionaries.

(10) The character string `[local]` (including the literal bracket characters) is *reserved* for local use. That is, no public dictionary will define a data name that includes this string. This allows experimentation with data items in a strictly local context, *i.e.* in cases where the CIF is not intended for interchange with any other user.

(11) Where CIFs including local data items are expected to enjoy a public circulation, authors may register a *reserved prefix* for their sole use. The registry is available on the web at http://www.iucr.org/iucr-top/cif/spec/reserved.html.

A reserved prefix, *e.g. foo*, must be used in the following ways:

(i) If the data file contains items defined in a DDL1 dictionary, the local data names assigned under the reserved prefix must contain it as their first component, *e.g.* `_foo_atom_site_my_item`.

(ii) If the data file contains items defined in a DDL2 dictionary, then the reserved prefix must be:

(*a*) the first component of data names in a category defined for local use, *e.g.* `_foo_my_category.my_item`.

(*b*) the first component following the period character in a data name describing a new item in a category already defined in a public dictionary, *e.g.* `_atom_site.foo_my_item`.

(12) There is no syntactic property identifying such a reserved prefix, so that software validating or otherwise handling such local data names must scan the entire registry and match registered prefixes against the indicated components of data names. Note that reserved prefixes may not themselves contain underscore characters.

2.2.7.4.6. *Note on handling of units*

(13) The published specification for CIF version 1.0 permitted data values expressed in different units to be tagged by variant data names (Hall *et al.*, 1991, p. 657):

> . . . Many numeric fields contain data for which the units must be known. Each CIF data item has a default units code which is stated in the CIF Dictionary. If a data item is not stored in the default units, the units code is appended to the data name. For example, the default units for a crystal cell dimension are ångströms. If it is necessary to include this data item in a CIF with the units of picometres, the data name of `_cell_length_a` is replaced by `_cell_length_a_pm`. Only those units defined in the CIF Dictionary are acceptable. The default units, except for the ångström, conform to the SI Standard adopted by the IUCr.

This approach is deprecated and has not been supported by any official CIF dictionary published subsequent to version 1.0 of the core. All data values must be expressed in the single unit assigned in the associated dictionary.

A small number of archived CIFs exist with variant data names as permitted by the above clause. If it is necessary to validate them against versions of the core dictionary subsequent to version 1.0, the formal compatibility dictionary cif_compat.dic (ftp://ftp.iucr.org/cifdics/cif_compat.dic) may be used for the purpose. *No other use should be made of this dictionary.*

2.2.7.4.7. *Data-value semantics*

(14) The STAR syntax permits retrieval of data by simply requesting a specific data name within a specific data block. Prior knowledge about data type (*e.g.* text or numbers), whether the item is looped or whether the item exists in the file at all is unnecessary. However, applications in general need to know data type, valid ranges of values and relationships between data items, and a program designer needs to know the purpose of the data item (*i.e.* what physical quantity or internal book-keeping function it represents). While such semantic information may be defined informally for local data items (ones not intended for exchange between different users or software applications), formal descriptions of the semantics associated with data values are catalogued in data dictionary files. Currently two formalisms (dictionary definition languages) for describing data-value attributes are supported; full specifications of these formalisms (known as DDL1 and DDL2) are provided in Chapters 2.5 and 2.6.

2.2.7.4.7.1. *Data typing*

(15) Four base data types are supported in CIF. These are:

(i) **numb**: a value interpretable as a decimal base number and supplied as an integer, a floating-point number or in scientific notation;

(ii) **char**: a value to be interpreted as character or text data (where the value contains white-space characters, it must be quoted);

(iii) **uchar**: a value to be interpreted as character or text data but in a case-insensitive manner (*i.e.* the values `FOO` and `foo` are to be taken as identical);

(iv) **null**: a special data type associated with items for which no definite value may be stored in computer memory. It is the type associated with the special character literal values `?` (query mark) and `.` (full point), which may appear as values for any data item within a data file (see Section 2.2.7.4.8 below). It is also the type assigned to items defined in dictionary files that may not occur in data files.

(16) *Comment:* Many applications distinguish between multi-line text fields and character-string values that fit within a single line of text. While this is a convenient practical distinction for coding purposes, formally both manifestations should be regarded as having the same base type, which might be 'char' or 'uchar'. Applications are at liberty to choose whether to define specific multi-line text subtypes, and whether to permit casting between subtypes of a base type. The examples of character-string delimiters in Section 2.2.7.1.4(20) are predicated on an approach that handles all subtypes of character or text data equivalently.

(17) Where the attributes of a data value are not available in a dictionary listing, it may be assumed that a character string inter-

pretable as a number should be taken to represent an item of type 'numb'. However, an explicit dictionary declaration of type will override such an assumption.

2.2.7.4.7.2. *Subtyping*

(18) The base data types detailed in the previous section are very general and need to be refined for practical application. Refinement of types is to some extent application-dependent, and different subtypes are supported for data items defined by DDL1 and DDL2 dictionary files. The following notes indicate some considerations, but the relevant dictionary files and documentation should be consulted in each case.

(19) *DDL1 dictionaries*. Values of type 'numb' may include a standard uncertainty in the final digit(s) of the number where the associated item definition includes the attribute

```
_type_conditions    esd
```

(or `_type_conditions su`, a synonym introduced to DDL1 in 2005). For example, a value of 34.5(12) means 34.5 with a standard uncertainty of 1.2; it may also be expressed in scientific notation as 3.45E1(12).

(20) *DDL2 dictionaries*. DDL2 provides a number of tags that may be used in a dictionary file to specify subtypes for data items defined by that dictionary alone. Examples of the subtypes specified for the macromolecular CIF dictionary are:

code	identifying code strings or single words
ucode	identifying code strings or single words (case-insensitive)
uchar1	single-character codes (case-insensitive)
uchar3	three-character codes (case-insensitive)
line	character strings forming a single line of text
uline	character strings forming a single line of text (case-insensitive)
text	multi-line text
int	integers
float	floating-point real numbers
yyyy-mm-dd	dates
symop	symmetry operations
any	any type permitted

2.2.7.4.8. *Special generic values*

(21) The unquoted character literals `?` (query mark) and `.` (full point) are special and are valid expressions for any data type.

(22) The value `?` means that the actual value of a requested data item is *unknown*.

(23) The value `.` means that the actual value of a requested data item is *inapplicable*. This is most commonly used in a looped list where a data value is required for syntactic integrity.

2.2.7.4.9. *Embedded data semantics*

(24) The attributes of data items defined in CIF dictionaries serve to direct crystallographic applications in the retrieval, storage and validation of relevant data. In principle, a CIF might include as data items suitably encoded fields representing data suitable for manipulation by text processing, image, spreadsheet, database or other applications. It would be useful to have a formal mechanism allowing a CIF to invoke appropriate content handlers for such data fields; this is under investigation for the next CIF version specification.

2.2.7.4.10. *CIF conventions for special characters in text*

(25) The one existing example of embedded semantics is the text character markup introduced in the CIF version 1.0 specification and summarized in paragraphs (30)–(37) below. The specification is silent on which fields should be interpreted according to these markup conventions, but the published examples suggest that they may be used in any character field in a CIF data file except as prohibited by a dictionary directive. It is intended that the next CIF version specification shall formally declare where such markup may be used.

2.2.7.4.11. *Handling of long lines*

(26) The restriction in line length within CIF requires techniques to handle without semantic loss the content of lines of text exceeding the limit (2048 characters in this revision, 80 characters in the initial CIF specification). The line-folding protocol defined here provides a general mechanism for wrapping lines of text within CIFs to any extent within the overall line-length limit. A specific application where this would be useful is the conversion of lines longer than 80 characters to the CIF version 1.0 limit. This 80-character limit is used in the examples below for illustrative purposes.

These techniques are applied only to the contents of text fields and to comments.

In order to permit such folding, a special semantics is defined for use of the backslash. It is important to understand that this does not change the syntax of CIF version 1.0. All existing CIFs conforming to the CIF version 1.0 specification can be viewed as having exactly the same semantics as they now have. Use of these transformational semantics is optional, but recommended.

In order to avoid confusion between CIFs that have undergone these transformations and those that have not, the special comment beginning with a hash mark immediately followed by a backslash (`#\`) as the last non-blank characters on a line is reserved to mark the beginning of comments created by folding long-line comments, and the special text field beginning with the sequence line termination, semicolon, backslash (`<eol>;\`) as the only non-blank characters on a line is reserved to mark the beginning of text fields created by folding long-line text fields.

The backslash character is used to fold long lines in character strings and comments. Consider a comment which extends beyond column 80. In order to provide a comment with the same meaning which can be fitted into 80-character lines, prefix the comment with the special comment consisting of a hash mark followed by a backslash (`#\`) and the line terminator. Then on new lines take appropriate fragments of the original comment, beginning each fragment with a hash mark and ending all but the last fragment with a backslash. In doing this conversion, check for an original line that ends with a backslash followed only by blanks or tabs. To preserve that backslash in the conversion, add another backslash after it. If the next lexical token (not counting blanks or tabs) is another comment, to avoid fusing this comment with the next comment, be sure to insert a line with just a hash mark.

Similarly, for a character string that extends beyond column 80,

(i) first convert it to be a text field delimited by line termination–semicolon (`<eol>;`) sequences,

(ii) then change the initial line termination–semicolon (`<eol>;`) sequence to line termination–semicolon–backslash–line termination (`<eol>;\<eol>`),

(iii) and break all subsequent lines that do not fit within 80 columns with a trailing backslash. In the course of doing the translation,

(*a*) check for any original text lines that end with a backslash followed only by blanks or tabs;

(*b*) to preserve that backslash in the conversion, add another backslash after it, and then an empty line.

(More formally, the line folding should be done separately and directly on single-line non-semicolon-delimited character strings to allow for recognition of the fact that no terminal line termination is intended – see below.)

In order to understand this scheme, suppose the CIF fragment (1) below were considered to have long lines. They could be transformed into (2) as follows:

(1) *Initial CIF*

```
#######################################################
### CIF submission form for Rietveld refinements
###
###                        Version 14 December 1998
###
#######################################################
data_znvodata
_chemical_name_systematic
; zinc dihydroxide divanadate dihydrate
;

_chemical_formula_moiety      ’H2 O9 V2 Zn3, 2(H2 O)’
_chemical_formula_sum         ’H6 O11 V2 Zn3’
_chemical_formula_weight      480.05
```

(2) *Transformed CIF*

```
#\
###########################\
###########################
### CIF submission form for Rietveld refinements
###
###                        Version 14 December 1998
###
#######################################################
data_znvodata
_chemical_name_systematic
;\
 zinc dihydroxide divan\
adate dihydrate
;

_chemical_formula_moiety
;\
H2 O9 V2 Zn3, 2(H2 O)\
;
_chemical_formula_sum         ’H6 O11 V2 Zn3’
_chemical_formula_weight      480.05
```

In making the transformation from the backslash-folded form to long lines, it is very important to strip trailing blanks before attempting to recognize a backslash as the last character. When reassembling text-field lines, no reassembly should be done except in text fields that begin with the special sequence described above, line termination–semicolon–backslash–line termination, (`<eol>;\<eol>`), so that text fields that happen to contain backslashes but which were not created by folding long lines are not changed. It is also important to remove the trailing backslashes when reassembling long lines. The final line termination–semicolon sequence of a text field takes priority over the reassembly process and ends it, but a trailing backslash on the last line of a text field very nicely conveys the information that no trailing line termination is intended to be included within the character string.

Similarly, when reassembling long-line comments, the reassembly begins with a comment of the form hash–backslash–line termination. The initial hash mark is retained and then a forward scan is made through line terminations and blanks for the next comment, from which the initial hash mark is stripped and then the contents of the comment are appended. If that comment ends with a backslash, the trailing backslash is stripped and the process repeats. Note that the process will be ended by intervening tags, values, data blocks or other non-white-space information, and that the process will not start at all without the special hash–backslash–line termination comment.

Since there are very few, if any, CIFs that contain text fields and comments beginning this way, in most cases it is reasonable to adopt the policy of doing this processing unless it is disabled.

Here is another example of folding. The following three text fields would be equivalent:

```
;C:\foldername\filename
;
```

```
;\
C:\foldername\filename
;
```

and

```
;\
C:\foldername\file\
name
;
```

but the following example would be a two-line value where the first line had the value `C:\foldername\file\` and the second had the value `name`:

```
;
C:\foldername\file\
name
;
```

Note that backslashes should not be used to fold lines outside of comments and text fields. That would introduce extraneous characters into the CIF and violate the basic syntax rules. In any case, such action is not necessary.

2.2.7.4.12. *Dictionary compliance*

(27) Dictionary files containing the definitions and attribute sets for the data items contained in a CIF should be identified within the CIF by some or all of the data items

```
_audit_conform_dict_name
_audit_conform_dict_version
_audit_conform_dict_location
```

corresponding to DDL1 dictionaries or

```
_audit_conform.dict_name
_audit_conform.dict_version
_audit_conform.dict_location
```

for DDL2 dictionaries. Where no such information is provided, it may be assumed that the file should conform against the core CIF dictionary.

(28) The **`_audit_conform`** data items may be looped in cases where more than one dictionary is used to define the items in a CIF and they may include dictionaries of local data items provided such dictionary files have been prepared in accordance with the rules of the appropriate DDL.

(29) A detailed protocol exists for locating, merging and overlaying multiple dictionary files (McMahon *et al.*, 2000) (see Section 3.1.9).

2.2.7.4.13. *CIF markup conventions*

(30) If permitted by the relevant dictionary and if no other indication is present, the contents of a text or character field are assumed to be interpretable as text in English or some other human language. Certain special codes are used to indicate special characters or accented letters not available in the ASCII character set, as listed below.

2.2.7.4.14. *Greek letters*

(31) In general, the corresponding letter of the Latin alphabet, prefixed by a backslash character. The complete set is:

α	Α	`\a`	`\A`	alpha	ν	Ν	`\n`	`\N`	nu
β	Β	`\b`	`\B`	beta	o	Ο	`\o`	`\O`	omicron
χ	Χ	`\c`	`\C`	chi	π	Π	`\p`	`\P`	pi
δ	Δ	`\d`	`\D`	delta	θ	Θ	`\q`	`\Q`	theta
ε	Ε	`\e`	`\E`	epsilon	ρ	Ρ	`\r`	`\R`	rho
φ	Φ	`\f`	`\F`	phi	σ	Σ	`\s`	`\S`	sigma
γ	Γ	`\g`	`\G`	gamma	τ	Τ	`\t`	`\T`	tau
η	Η	`\h`	`\H`	eta	υ	Υ	`\u`	`\U`	upsilon
ι	Ι	`\i`	`\I`	iota	ω	Ω	`\w`	`\W`	omega
κ	Κ	`\k`	`\K`	kappa	ξ	Ξ	`\x`	`\X`	xi
λ	Λ	`\l`	`\L`	lambda	ψ	Ψ	`\y`	`\Y`	psi
μ	Μ	`\m`	`\M`	mu	ζ	Ζ	`\z`	`\Z`	zeta

2.2.7.4.15. *Accented letters*

(32) Accents should be indicated by using the following codes before the letter to be modified (*i.e.* use `\'e` for an acute e):

`\'`	acute (é)	`\"`	umlaut (ü)
`\=`	overbar or macron (ā)	`` \` ``	grave (à)
`\~`	tilde (ñ)	`\.`	overdot (ȯ)
`\^`	circumflex (â)	`\;`	ogonek (ų)
`\<`	hacek or caron (ǒ)	`\,`	cedilla (ç)
`\>`	Hungarian umlaut or double accented (ő)	`\(`	breve (ŏ)

These codes will always be followed by an alphabetic character.

2.2.7.4.16. *Other characters*

(33) Other special alphabetic characters should be indicated as follows:

`\%a`	a-ring (å)	`\?i`	dotless i (ı)	`\&s`	German ‘ss’ (ß)
`\/o`	o-slash (ø)	`\/l`	Polish l (ł)	`\/d`	barred d (đ)

Capital letters may also be used in these codes, so an ångström symbol (Å) may be given as `\%A`.

(34) Superscripts and subscripts should be indicated by bracketing relevant characters with circumflex or tilde characters, thus:

superscripts	`Csp^3^`	for	Csp^3
subscripts	`U~eq~`	for	U_{eq}

The closing symbol is essential to return to normal text.

(35) Some other codes are accepted by convention. These are:

`\%`	degree (°)	`\\times`	×
`--`	dash	`+-`	±
`---`	single bond	`-+`	∓
`\\db`	double bond	`\\square`	□
`\\tb`	triple bond	`\\neq`	≠
`\\ddb`	delocalized double bond	`\\rangle`	>
`\\sim`	∼	`\\langle`	<
(*Note:* `~` is the code for subscript)		`\\rightarrow`	→
`\\simeq`	≃	`\\leftarrow`	←
`\\infty`	∞		

Note that `\\db`, `\\tb` and `\\ddb` should always be followed by a space, *e.g.* C=C is denoted by `C\\db C`.

2.2.7.4.17. *Typographic style codes*

(36) The codes indicated above are designed to refer to special characters not expressible within the CIF character set, and the initial specification did not permit markup for typographic style such as italic or bold-face type. However, in some cases the ability to indicate type style is useful, and in addition to the codes above HTML-like conventions are allowed of surrounding text by `<i> </i>` to indicate the beginning and end of italic, and by `<b> </b>` to indicate the beginning and end of bold-face type.

(37) If it is necessary to convey more complex typographic information than is permitted by these special character codes and conventions, the entire text field should be of a richer content type allowing detailed typographic markup.

References

Bernstein, H. J. (2002). *Some comments on parsing for computer programming languages.* http://www.bernstein-plus-sons.com/TMM/Parsing.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. (1999). *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. Network Working Group. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Freed, N. & Borenstein, N. (1996). *Multipurpose Internet Mail Extensions (MIME) Part two: media types.* RFC 2046. Network Working Group. http://www.ietf.org/rfc/rfc2046.txt.

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

McMahon, B., Westbrook, J. D. & Bernstein, H. J. (2000). *Report of the COMCIFS Working Group on Dictionary Maintenance.* http://www.iucr.org/iucr-top/cif/spec/dictionaries/maintenance.html.

Ulrich, E. L. *et al.* (1998). XVIIth Intl Conf. Magn. Res. Biol. Systems. Tokyo, Japan.

Westbrook, J. D. & Hall, S. R. (1995). *A dictionary description language for macromolecular structure.* http://ndbserver.rutgers.edu/mmcif/ddl/.