2.2

Specification of the Crystallographic Information File (CIF)

By J. C. Bollinger and J. R. Hester

2.2.1 Introduction

Scientific investigation furthers general human knowledge only to the extent that the details of experiments performed, results obtained, and conclusions drawn are recorded and communicated. Since the advent of electronic computers, the focus for recording and communicating scientific data has ever increasingly been on electronic, machine-actionable data representations and electronic means of data exchange. Yet even within a single scientific domain, the experimental techniques employed, the characteristics of the equipment used and of the resulting data, and even the subject and nature of the conclusions drawn varies over time, researcher, and experimental purpose. As a result, a vast diversity electronic data formats and associated software serving various specific purposes and domains has been introduced and used – and, in many cases, has subsequently faded to obscurity. Such a diversity of data formats and software exists in crystallography, but here, ongoing efforts over the past 30 years have yielded the widely accepted and used general framework for crystallographic information described in this volume. This Crystallographic Information Framework (CIF) comprises a generic, flexible, composable data model, a large and growing shared data ontology, formats for externalizing data for storage and transmission, and both special- and general-purpose software for producing, consuming, and manipulating these data. All of this began with the the introduction of the Crystallographic Information File (also and originally CIF), and it is on this, the file format, that this chapter focuses.

The earliest version of CIF (Hall, et al., 1991) was introduced as an application profile of the STAR format (chapter 2.1) with some features omitted, a few domain-specific conventions defined, and a moderate vocabulary of data identifiers introduced. This version of the file format, retrospectively designated CIF 1.0, was replaced in 2003 by a minor revision, CIF 1.1 (COMCIFS, 2003), that was subsequently documented in the previous edition of this volume. These versions of the file format have served their purpose well, and CIF 1.1 continues to do so today. Nevertheless, its limitations with respect to the text and symbols it can directly represent and the unwieldy mechanisms it requires for representing complex data structures prompted a more substantial revision addressing these shortcomings, published in 2016 as CIF 2.0 (Bernstein, et al, 2016). The CIF 1.1 and CIF 2.0 file formats are both described in this chapter. They feature order-independence and flexible layout, are without any fixed requirement on which data or categories of data are presented within, and place minimal limitations on the introduction of new data items. They can represent rich data types and

Affiliations: John C. Bollinger, Saint Jude Children's Research Hospital, Department of Structural Biology, 262 Danny Thomas Place, Memphis, Tennessee 38105, USA; James R. Hester, Australian Nuclear Science and Technology Organisation, New Illawarra Road, Lucas Heights, NSW 2234, Australia.

complex relational data, especially in conjunction with relationally-oriented dictionaries such as mmCIF (chapter XXX) and dictionary definition languages such as DDL2 and DDLm (chapters XXX and YYY).

The balance of this chapter begins with a description of the data model underpinning CIF data representations and data processing, which additionally supplies standard terms for many of the CIF entities and constructs to be discussed. It proceeds to discuss CIF file syntax for representing such data, with complete, formal definitions of both version 1.l and 2.0 syntax, and some commentary on how these relate to older CIF usage. The relationship of each CIF syntax version to its parent STAR version is explained, and the differences between the two CIF versions are described in detail. There follows a discussion of aspects of CIF semantics that apply broadly to interpreting CIF documents and the data within, and the chapter concludes with some considerations for using CIF in data exchange and data archiving roles.

2.2.2 Data model and terminology

A “data model” is an abstract model describing individual elements of data and how they are structured and related. A simple data model underpins all expressions of CIF data and all forms of CIF processing, and a summary of that model and the corresponding terminology used throughout this chapter is presented below. Detailed specifications for two syntaxes for expressing data conforming to this model are presented in §2.2.3. A schematic representation of this model is presented in Figure 1.

(1) A **data value** is a coherent unit of of information. It may represent a number, a word, extended discursive text, an image, or substantially any object that affords a finite electronic representation. Data typing is discussed more fully in section 2.2.4.2.2

(2) A **data name** or **tag** is a textual identifier for one or more data values. Following STAR, CIF data names start with an underscore character (_).

(3) A **data item** is a data value together with its corresponding data name.

(4) A **looped list** or **loop** is a non-empty set of sets of correlated data items (**packets**), characterized by a shared, non-empty set of data names. Each packet contains exactly one item for each of its loop's data names. Loops are thus analogous to relational tables.

(5) A **save frame** is a container for data items, identified by a **frame code**. No data name is repeated within the same save frame. Multiple items for the same data names may be presented without repeating data names by use of looped lists.

(6) A **data block** is a container for data items, identified by a **block code**. Data blocks can have the same data item content as save frames, with the same name uniqueness requirement, but they can also contain save frames. No frame code is repeated within the same data block. Data name uniqueness requirements associated with a data block apply only to the names directly within – they do not extend to the contents of any save frames.

(7) A **CIF** is a collection of zero or more data blocks. Data block codes are not repeated within the same CIF.

CIF identifiers – data names, frame codes, and block codes – are case-insensitive. Specifically, two such entities are equivalent for CIF's purposes if their names are canonical caseless matches of each other according to the Unicode canonical caseless matching algorithm (The Unicode Consortium, 2014*b*). Substituting one identifier for another that is a canonical caseless match to it does not change the meaning or validity of a CIF data set, and two such equivalent identifiers may be used interchangeably in native CIF documents, provided that each is individually permitted by the applicable CIF syntax.

2.2.3 CIF Syntax

Two STAR-based syntaxes for representing CIF data are presented in this Chapter: CIF 1.1 and CIF 2.0. These are similar in form, but not wholly interoperable. A full description of each syntax is presented in the following sections, followed by a comparison of the two.

Where specific characters are discussed, Unicode formalism and character codes of the general form ***U+[[x]x]xxxx*** are employed. That is, the prefix "U+" followed by the code point assigned to the character by Unicode as a four- to six-digit hexadecimal number, with leading zeroes as necessary to pad the numeric part to at least four digits.

Each language description comprises both lexical and syntactic grammar, presented via a formalism based on the ISO 14977 Extended Backus-Naur Form (EBNF) standard (International Standards Organization, 1996). This permits precise formal specification of **productions** defining how terminal symbols – sequences of one or more specific characters – can be assembled into aggregates represented by non-terminal symbols, those further assembled into larger aggregates, and so forth, ultimately to achieve an aggregate corresponding to an entire CIF. Among the aggregates assembled along the way are some corresponding to the entities described in the data model presented in section 2.2.2. A summary of the formalism is presented in Table 1.

Table 1: EBNF summary

EBNF	Role	Example	Notes
`'...'` `"..."`	terminal symbol	`'#\#CIF_1.1'` `"'"`	Matches the text enclosed by the quotes / apostrophes

`?...?`	terminal symbol[1]	`?U+0009?`
		`?U+0021 - U+007E?`
		`?EOL?`
			Matches one Unicode character having the specified code point or a code point in the specified range
			Matches the system-specific text end-of-line

`,`	concatenation	`x, y`	Matches a sequence that matches `x` followed by one that matches `y`

1 The `?...?` is a "special sequence", which is an extension point provided by ISO 14977. These have application-defined meaning, thus it is these specifications, not the ISO standard, that define the meaning of the special sequences used within.

`|`	alternation	`x | y`	Matches a sequence that matches `x` or a sequence that matches `y`
`-`	difference	`x - y`	Matches a sequence that matches `x` but does not match `y`
`(...)`	simple grouping	`( x )`	Matches a sequence that matches `x`
`[...]`	optional grouping	`[ x ]`	Matches either a sequence that matches `x` or an empty sequence
`{...}`	repeatable grouping	`{ x }`	Matches zero or more sequences that each match `x`
n`*`	repetition count	`2049*x`	Matches the specified exact number of sequences that each match `x`
`;`	terminator	`;`	Marks the end of a production

2.2.3.1 CIF 1.1 syntax

CIF 1.1's native syntax is a strict subset of STAR 1.0 (Hall, 1991; Hall & Spadaccini, 1994). Notwithstanding its identification of characters in terms of ASCII and Unicode, CIF 1.1 relies on local text conventions for character encoding and other details. In particular, CIF 1.1 is characterized in terms of lines of text, where the definition of what constitutes such a line is left to local convention. However a line of text is characterized or characters are encoded in a given local environment, CIF 1.1 limits the length of its lines to 2048 characters.

The formal grammar presented in this section therefore requires a mechanism by which to represent the end of a line of text, independent of any particular convention. The EBNF special sequence *`?EOL?`* is chosen for this purpose. The formal grammar is predicated on that special sequence matching the end of each logical line, including the last. Where local convention employs one or more sequences of one or more input characters to terminate or separate lines, *`?EOL?`* matches each such sequence. Where local convention employs some other mechanism, such as logical record or file boundaries, *`?EOL?`* represents the end of a line without matching any input characters. In particular, for the purpose of these syntax specifications, the last line of a CIF is always terminated by *`?EOL?`*, regardless of local convention or the file's structure and physical content.

It is essential to account for the textual character of CIF 1.1 documents when they are transferred from one computer system to another, lest their meaning be changed by reinterpretation according to different text conventions.

2.2.3.1.1 CIF 1.1 character set

CIF 1.1 files consist of lines of printable characters from the ASCII character set (International Standards Organization, 1991) (***U+0020*** through ***U+007E***), plus non-printable characters horizontal tab (***U+0009***), line feed (***U+000A***), and carriage return (***U+000D***). Some of these are reserved or have special significance in certain CIF contexts, but others are ordinary, never serving any purpose except to represent themselves.

```
cif1-char =  cif1-wspace-char | cif1-non-blank-char ;

cif1-non-blank-char = cif1-ordinary-char | '"' | '#' | '$' | "'" | '_' | ';'
   | '[' | ']' ;

cif1-wspace-char = ?U+0009? | ?U+000A? | ?U+000D? | ' ' ;

cif1-ordinary-char = '!' | '%' | '&' | '(' | ')' | '*' | '+' | ',' | '-' | '.'
   | '/' | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':'
   | '<' | '=' | '>' | '?' | '@' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
   | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S'
   | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '\' | '^' | '`' | 'a' | 'b'
   | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n'
   | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
   | '{' | '|' | '}' | '~' ;
```

Some syntax descriptions predating the CIF 1.1 specifications permitted additional non-printing characters, but only the characters designated above are permitted in CIF 1.1 documents. Some of the designated control characters or combinations of them may be recognized as line terminators according to local convention; this affects the interpretation of documents containing them, but not whether they are permitted to appear.

2.2.3.1.2 Inline text

Several CIF 1.1 constructs are forbidden to span lines. In support of these, syntactic constructs are defined to represent sequences of characters that do not span lines or contain a line terminator – one for zero or more characters of general text, and one for non-empty white space runs.

```
cif1-inline-chars = { cif1-char } - cif1-line-span ;

cif1-inline-wspace = ( cif1-wspace-char, { cif1-wspace-char } )
   - cif1-line-span ;
```

Both are defined in part relative to character sequences that do span lines.

```
cif1-line-span = { cif1-char }, ?EOL?, { cif1-char | ?EOL? } ;
```

The environment-dependence of line termination semantics interferes with writing strictly constructive productions for such runs of inline text.

2.2.3.1.3 White space and comments

The space, horizontal tab, carriage return, and line feed characters serve as white space for separating CIF 1.1 syntactic constructs. Where not individually or jointly significant for line termination, they serve as in-line whitespace.

Additionally, CIF 1.1 provides for comments introduced by a '#' character and continuing to the end of the line.

```
cif1-comment = '#', cif1-inline-chars ;
```

This definition of a comment does not incorporate the *?EOL?* that must follow. That allows for the end-of-line to be attributed instead to the starting delimiter of an immediately-following text field (see §2.2.3.1.5), and overall requires that the needed *?EOL?* be ensured by the symbol's context wherever it appears.

Example:

```
# This is a comment, ending at the full point.
```

Because of CIF's sensitivity to line boundaries, the productions for some of the constructs to be presented later will be simplified by the introduction of constructs for additional varieties of whitespace run. The usual syntactic separator is a nonempty run of whitespace characters and possibly comments, neither beginning nor ending with a comment, possibly spanning multiple lines:

```
cif1-wspace = ( cif1-inline-wspace | ?EOL? ), cif1-wspace-any;

cif1-wspace-any = { cif1-wspace-char | ( [ cif1-comment ], ?EOL?) } ;
```

The symbol *cif1-wspace* represents what this chapter means by the unqualified term “whitespace” as it applies to CIF 1.1 syntax. Some contexts, however, require more specifically a run of whitespace and comments that ends with the end of a line:

```
cif1-wspace-lines = [ cif1-inline-wspace, [ cif1-comment ] ], ?EOL?,
   { cif1-wspace-to-eol } ;

cif1-wspace-to-eol = [ cif1-inline-wspace ], [ cif1-comment ], ?EOL? ;
```

2.2.3.1.4 Reserved character sequences

CIF 1.1 has a small number of reserved character sequences. All of them are case-insensitive, as their productions demonstrate. Some of them have syntactic significance in CIF 1.1 when they appear unquoted (`data_` , `loop_` , and `save_`):

```
cif1-data-token = ( 'D' | 'd' ), ( 'A' | 'a' ), ( 'T' | 't' ), ( 'A' | 'a' ),
   '_' ;

cif1-save-token = ( 'S' | 's' ), ( 'A' | 'a' ), ( 'V' | 'v' ), ( 'E' | 'e' ),
'_' ;

cif1-loop-token = ( 'L' | 'l' ), ( 'O' | 'o' ),( 'O' | 'o' ), ( 'P' | 'p' ) ,
   '_' ;
```

Others are STAR keywords that are reserved for possible future use in CIF (`global_` and `stop_`):

```
cif1-global-token = ( 'G' | 'g' ), ( 'L' | 'l' ), ( 'O' | 'o' ), ( 'B' | 'b' ),
   ( 'A' | 'a' ), ( 'L' | 'l' ), '_' ;

cif1-stop-token = ( 'S' | 's' ), ( 'T' | 't' ), ( 'O' | 'o' ), ( 'P' | 'p' ),
   '_' ;
```

Additionally, whitespace-delimited data values (next section) beginning with a dollar sign ($), opening square bracket ([), or closing square bracket (]) character are reserved, and must not appear in CIF 1.1

documents.

2.2.3.1.5 Data values

CIF 1.1 data values are presented as sequences of characters in one of three forms. They may be whitespace-delimited strings, whose allowed form varies depending on whether they appear at the start of a line:

```
cif1-wsdelim-string = cif1-wsdelim-string-sol
   | ( ';', { cif1-non-blank-char } ) ;

cif1-wsdelim-string-sol = ( cif1-ordinary-char, { cif1-non-blank-char } )
   - ( ( ( cif1-data-token | cif1-save-token ), { cif1-non-blank-char } )
         | cif1-loop-token | cif1-global-token | cif1-stop-token ) ;
```

They may be apostrophe- or quotation-mark-delimited strings. These may contain their delimiter as if it were an ordinary character, provided that it is not followed by a whitespace character, but they may not span lines.

```
cif1-quoted-string = ( cif1-quote-delim, cif1-quote-content,
   cif1-quote-delim )  | ( cif1-apos-delim, cif1-apos-content, cif1-apos-delim
   ) ;

cif1-quote-content = cif1-inline-chars - ( cif1-inline-chars,
   cif1-quote-delim, cif1-wspace-char, cif1-inline-chars ) ;

cif1-quote-delim = '"' ;

cif1-apos-content = cif1-inline-chars - ( cif1-inline-chars, cif1-apos-delim,
   cif1-wspace-char, cif1-inline-chars ) ;

cif1-apos-delim = "'" ;
```

Examples:

```
Bragg
12.345(6)
"reciprocal lattice"
'O'Malley & Smith'
```

They may also be **text fields** possibly spanning multiple lines, delimited by semicolon (;) characters appearing as the first characters of their lines.

```
cif1-text-field = cif1-text-delim, cif1-text-content, cif1-text-delim ;

cif1-text-content = { cif1-char | ?EOL? } - ( { cif1-char }, cif1-text-delim, {
   cif1-char } ) ;

cif1-text-delim = ?EOL?, ';' ;
```

Example:

```
;Multiple
lines
;
```

Whether a value is presented in delimited or undelimited form is significant in CIF 1.1, but the variety of delimiter is not, except in the sense that certain values, such as those spanning more than one line, can be represented only via a subset of defined forms. In particular, text field semantics mirror the above syntactic characterization in that the *?EOL?* of a text field's trailing delimiter is not considered part of the field's content.

Whatever the form in which it is presented, the content of a data value is case sensitive. As far as CIF itself is concerned, changing any character to a different one, no matter how the two are related, produces a distinct value. Of course, applications consuming the data are not constrained to observe the same distinction.

Text fields have implicit leading whitespace because their delimiting semicolons must appear as the first character of a line. All other data values must be separated from the preceding data name or data value by whitespace, or from a preceding comment by at least an end-of-line:

```
cif1-wspace-data-value = ( cif1-wspace, cif1-quoted-string )
   | ( [ cif1-wspace-lines ], cif1-inline-wspace, cif1-wsdelim-string )
   | ( cif1-wspace-lines, cif1-wsdelim-string-sol )
   | ( [ cif1-wspace ], [ cif1-comment ], cif1-text-field ) ;
```

2.2.3.1.6 Data names

CIF data names begin with an underscore character and may contain any non-whitespace character, but CIF 1.1 forbids that they be longer than 75 characters overall:

```
cif1-data-name = ( '_' , cif1-non-blank-char, { cif1-non-blank-char } )
   - cif1-overlength-identifier ;

cif1-overlength-identifier = 76 * cif1-non-blank-char, { cif1-non-blank-char }
   ;
```

Examples:

```
_DDL1_style
_DDL2.style
_DDLm.style
```

Data names are case-insensitive. Transforming one by changing the alphabetic case of one or more of its letters neither changes its significance nor is relevant to data blocks' and save frames' data-name uniqueness requirements.

2.2.3.1.7 Data items

CIF 1.1 data values can be associated with data names to form data items in two ways. A single value can be paired with a single name immediately preceding it to form one data item, or one or more lists of one or more correlated data values can be be associated with one list of data names in a **loop** construct to form multiple items, one for each value:

```
cif1-data = cif1-scalar-item | cif1-looped-list ;
```

```
cif1-scalar-item =  cif1-data-name, cif1-wspace-data-value ;

cif1-looped-list = cif1-loop-token, cif1-wspace, cif1-data-name,
   { cif1-wspace, cif1-data-name }, cif1-wspace-data-value,
   { cif1-wspace-data-value } ;
```

Examples:

```
_chemical_formula_sum 'C10 H6 O2'

loop_
_atom_site.label
_atom_site.type_symbol
_atom_site.fract_x
_atom_site.fract_y
_atom_site.fract_z
C1 C  0.0251(4) 0.7811(2)   -0.00226(16)
C2 C -0.1785(4) 0.6637(2)   -0.07803(15)
O3 O -0.2912(3) 0.69231(14) -0.17651(13)
```

The values in a loop are logically divided into lists of adjacent values numbering the same as the loop's list of data names, and each value in each list is associated with the data name at the corresponding position in the data name list to form lists of correlated data items (packets). CIF 1.1 requires that the number of data names in a looped list evenly divides the number of values in it, though this constraint cannot be expressed in the EBNF formalism with which the syntax is herein presented.

It is common in CIF documents for the data values in a looped list to be presented arranged in rows corresponding to the value lists described above. It should be emphasized that such an arrangement is not significant to CIF; it serves only to aid human readers, and if done incorrectly it can instead deceive them. Only the overall sequence of the data values is significant for associating data values with each other and with data names.

2.2.3.1.8 Data blocks and save frames

CIF 1.1 data items are contained within data blocks and save frames. These are similar: each has a distinguishing identifier, and each establishes the scope for a uniqueness requirement on the data names appearing directly within. Each data name presented must be unique, in the case-insensitive sense discussed previously, among the data names presented in its immediately-containing data block or save frame.

The beginning of a data block is marked by a data block heading consisting of the concatenation of a five-character sequence case-insensitively matching “data_” with the one or more non-whitespace characters of that block's block code.

Example:

```
data_sj13_025
```

The end of a data block is implicit at the beginning of the next data block or the end of the file, whichever comes first. Data blocks do not nest.

Data blocks may contain save frames, interspersed with data in any sequence. As CIF 1.1 has been used to date, save frames appear only in dictionary files, not in data files; these types of file are not otherwise distinguished by syntax. It is permissible for a CIF processor to consider the unexpected appearance of a save frame in a data block to be erroneous. The beginning of a save frame is marked by a save frame heading consisting of the concatenation of a five-character sequence case-insensitively matching “save_” with the one or more non-whitespace characters of that frame's frame code. The end is marked by a five-character sequence case-insensitively matching “save_”, without any frame code. Save frames do not nest.

```
cif1-data-block = cif1-data-heading, { cif1-block-content } ;

cif1-data-heading = cif1-data-token,
   ( cif1-container-code - cif1-overlength-identifier ) ;

cif1-block-content = cif1-wspace, ( cif1-data | cif1-save-frame ) ;

cif1-save-frame = cif1-save-heading, { cif1-frame-content }, cif1-wspace,
   cif1-save-token ;

cif1-save-heading = cif1-save-token,
   ( cif1-container-code - cif1-overlength-identifier ) ;

cif1-frame-content = cif1-wspace, cif1-data ;

cif1-container-code = ( cif1-non-blank-char, { cif1-non-blank-char } ) ;
```

Example:

```
data_example
_section '2.2.3.1.8'

save_a_amino_acids
  loop_
    _aa.name
    _aa.3_character_symbol
    _aa.1_character_symbol
    alanine             Ala   A
    arginine            Arg   R
    asparagine          Asn   N
    'aspartic acid'     Asp   D
save_

# still in data block "example"
_example.version 1
```

CIF 1.1 requires that block codes and frame codes not exceed 75 characters in length. All save frames in the same data block must have distinct frame codes, as evaluated in the case-insensitive sense previously discussed.

2.2.3.1.9 CIF 1.1 file

An overall CIF 1.1 document begins with an optional leading code identifying the syntax version, which, if present, has a form that can also be interpreted as an ordinary comment by processors that do

not care to look for it. The tail of that line can contain arbitrary text as long as the version code is separated from it by whitespace. No significance is attributed to such text, and it is rarely provided in practice, so a CIF-version comment for CIF 1.1 is normally expressed simply as

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

Additional leading whitespace is permitted, followed by a sequence of zero or more data blocks. All data blocks in the same CIF must have distinct block codes, as evaluated in the case-insensitive sense previously discussed. No sequences of 2049 or more characters that fail to span at least two lines may appear, with line terminators not counting toward this limit.

```
cif1-cif = ( [ cif1-version-code, [ cif1-inline-wspace,
   [ cif1-inline-chars ] ], ?EOL? ], cif1-wspace-any, [ cif1-data-blocks ] ] )
   - ( { cif1-char }, ( 2049 * cif1-char ) - cif1-line-span, { cif1-char } ) ;

cif1-version-code = '#\#CIF_1.1' ;

cif1-data-blocks = cif1-data-block, { cif1-wspace, cif1-data-block },
   [ cif1-wspace ] ;
```

CIF 1.1 places no minimum requirement on data content. In particular, an empty document or one containing only whitespace is syntactically valid. Figure 2 presents a simple CIF expressed using CIF 1.1 syntax.

2.2.3.1.10 Comparison with STAR 1.0

CIF 1.1 native syntax is a strict subset of STAR 1.0 syntax. CIF has these restrictions relative to STAR:

- CIF 1.1 imposes a 2048-character limit on line lengths, whereas STAR does not limit line lengths.
- CIF 1.1 limits block codes, frame codes, and data names to 75 characters, whereas STAR does not limit these.
- CIF 1.1 requires case-insensitive uniqueness of block codes, frame codes, and data names within their scopes, whereas STAR requires only exact uniqueness.
- CIF 1.1 excludes the vertical tab and form feed characters from its allowed character set.
- CIF 1.1 does not permit nested loops.
- CIF 1.1 documents may not contain STAR `global_` sections.
- CIF 1.1 reserves the use of character sequences having the form of STAR frame references as whitespace-delimited data values.

2.2.3.1.11 Complete formal grammar

The CIF 1.1 formal syntax and grammar were presented piecemeal in preceding sections. This section

presents the full set of productions in conventional top-down order.

```
cif1-cif = ( [ cif1-version-code, [ cif1-inline-wspace,
   [ cif1-inline-chars ] ], ?EOL? ], cif1-wspace-any, [ cif1-data-blocks ] ] )
   - ( { cif1-char }, ( 2049 * cif1-char ) - cif1-line-span, { cif1-char } ) ;

cif1-version-code = '#\#CIF_1.1' ;

cif1-data-blocks = cif1-data-block, { cif1-wspace, cif1-data-block },
   [ cif1-wspace ] ;

cif1-data-block = cif1-data-heading, { cif1-block-content } ;

cif1-data-heading = cif1-data-token,
   ( cif1-container-code - cif1-overlength-identifier ) ;

cif1-block-content = cif1-wspace, ( cif1-data | cif1-save-frame ) ;

cif1-save-frame = cif1-save-heading, { cif1-frame-content }, cif1-wspace,
   cif1-save-token ;

cif1-save-heading = cif1-save-token,
   ( cif1-container-code - cif1-overlength-identifier ) ;

cif1-frame-content = cif1-wspace, cif1-data ;

cif1-container-code = ( cif1-non-blank-char, { cif1-non-blank-char } ) ;

cif1-data = cif1-scalar-item | cif1-looped-list ;

cif1-scalar-item =  cif1-data-name, cif1-wspace-data-value ;

cif1-looped-list = cif1-loop-token, cif1-wspace, cif1-data-name,
   { cif1-wspace, cif1-data-name }, cif1-wspace-data-value,
   { cif1-wspace-data-value } ;

cif1-data-name = ( '_' , cif1-non-blank-char, { cif1-non-blank-char } )
   - cif1-overlength-identifier ;

cif1-overlength-identifier = 76 * cif1-non-blank-char, { cif1-non-blank-char }
   ;

cif1-wspace-data-value = ( cif1-wspace, cif1-quoted-string )
   | ( [ cif1-wspace-lines ], cif1-inline-wspace, cif1-wsdelim-string )
   | ( cif1-wspace-lines, cif1-wsdelim-string-sol )
   | ( [ cif1-wspace ], [ cif1-comment ], cif1-text-field ) ;

cif1-wsdelim-string = cif1-wsdelim-string-sol
   | ( ';', { cif1-non-blank-char } ) ;

cif1-wsdelim-string-sol = ( cif1-ordinary-char, { cif1-non-blank-char } )
   - ( ( ( cif1-data-token | cif1-save-token ), { cif1-non-blank-char } )
         | cif1-loop-token | cif1-global-token | cif1-stop-token ) ;
```

```
cif1-quoted-string = ( cif1-quote-delim, cif1-quote-content,
   cif1-quote-delim )  | ( cif1-apos-delim, cif1-apos-content, cif1-apos-delim
   ) ;

cif1-quote-content = cif1-inline-chars - ( cif1-inline-chars,
   cif1-quote-delim, cif1-wspace-char, cif1-inline-chars ) ;

cif1-quote-delim = '"' ;

cif1-apos-content = cif1-inline-chars - ( cif1-inline-chars, cif1-apos-delim,
   cif1-wspace-char, cif1-inline-chars ) ;

cif1-apos-delim = "'" ;

cif1-text-field = cif1-text-delim, cif1-text-content, cif1-text-delim ;

cif1-text-content = { cif1-char | ?EOL? } - ( { cif1-char }, cif1-text-delim, {
   cif1-char } ) ;

cif1-text-delim = ?EOL?, ';' ;

cif1-data-token = ( 'D' | 'd' ), ( 'A' | 'a' ), ( 'T' | 't' ), ( 'A' | 'a' ),
   '_';

cif1-save-token = ( 'S' | 's' ), ( 'A' | 'a' ), ( 'V' | 'v' ), ( 'E' | 'e' ),
'_';

cif1-loop-token = ( 'L' | 'l' ), ( 'O' | 'o' ),( 'O' | 'o' ), ( 'P' | 'p' ) ,
   '_' ;

cif1-global-token = ( 'G' | 'g' ), ( 'L' | 'l' ), ( 'O' | 'o' ), ( 'B' | 'b' ),
   ( 'A' | 'a' ), ( 'L' | 'l' ), '_' ;

cif1-stop-token = ( 'S' | 's' ), ( 'T' | 't' ), ( 'O' | 'o' ), ( 'P' | 'p' ),
   '_' ;

cif1-wspace = ( cif1-inline-wspace | ?EOL? ), cif1-wspace-any;

cif1-wspace-any = { cif1-wspace-char | ( [ cif1-comment ], ?EOL?) } ;

cif1-wspace-lines = [ cif1-inline-wspace, [ cif1-comment ] ], ?EOL?,
   { cif1-wspace-to-eol } ;

cif1-wspace-to-eol = [ cif1-inline-wspace ], [ cif1-comment ], ?EOL? ;

cif1-comment = '#', cif1-inline-chars ;

cif1-inline-chars = { cif1-char } - cif1-line-span ;

cif1-inline-wspace = ( cif1-wspace-char, { cif1-wspace-char } )
   - cif1-line-span ;

cif1-line-span = { cif1-char }, ?EOL?, { cif1-char | ?EOL? } ;

cif1-char =  cif1-wspace-char | cif1-non-blank-char ;

cif1-non-blank-char = cif1-ordinary-char | '"' | '#' | '$' | "'" | '_' | ';'
   | '[' | ']' ;
```

```
cif1-wspace-char = ?U+0009? | ?U+000A? | ?U+000D? | ' ' ;

cif1-ordinary-char = '!' | '%' | '&' | '(' | ')' | '*' | '+' | ',' | '-' | '.'
   | '/' | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':'
   | '<' | '=' | '>' | '?' | '@' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
   | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S'
   | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '\' | '^' | '`' | 'a' | 'b'
   | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n'
   | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
   | '{' | '|' | '}' | '~' ;
```

2.2.3.2 CIF 2.0 syntax

CIF 2.0's native syntax is derived from a subset of STAR 2.0 format (Spadaccini & Hall, 2012), but it deviates in some details (§2.2.3.2.10). Like STAR 2.0, CIF 2.0 is defined in terms of a sequence of Unicode characters, encoded via UTF-8 (The Unicode Consortium, 2014*a*). Unlike CIF 1.1, it is independent of local conventions for text files, thus it should be considered a binary format, although on many systems CIF 2.0 documents will nevertheless be substantially similar to conventional text documents. The remainder of this section describes the syntax of CIF 2.0 documents.

2.2.3.2.1 CIF 2.0 character set

The CIF 2.0 character set consists of all Unicode characters. More precisely, it consists of all characters corresponding to Unicode code points that are neither surrogate code points nor designated not-a-character code points, excepting C0 and C1 control characters other than horizontal tab, carriage return, and line feed. Characters in a Unicode private use area or associated with code points not yet assigned by the Unicode Consortium are expressly permitted by the CIF 2.0 syntax.

Such broad latitude in character use is possible in part because despite relying on the Unicode character set, CIF 2.0 itself does not observe full Unicode text semantics. Only a few of those characters that Unicode considers whitespace or to which it attributes significance for line termination are significant for those purposes in CIF 2.0 syntax: the space and horizontal tab characters serve the same whitespace roles in CIF 2.0 as they do in CIF 1.1, and the line feed and carriage return characters as well as carriage return / line feed pairs serve (any and all of them) as line terminators. Carriage return / line feed pairs are always considered a single line terminator, not two.

```
cif2-char = cif2-inline-char | ?U+000A? | ?U+000D? ;

cif2-line-term = ( ?U+000D?, [ ?U+000A? ] ) | ?U+000A? ;

cif2-inline-char = cif2-non-blank-char | cif2-blank-char ;

cif2-blank-char = ?U+0020? | ?U+0009? ;
```

```
cif2-non-blank-char = ?U+0021 - U+007E? | ?U+00A0 – U+D7FF?
   | ?U+E000 - U+FDCF?   | ?U+FDF0 - U+FEFE?   | ?U+FF00 - U+FFFD?
   | ?U+10000 - U+1FFFD? | ?U+20000 - U+2FFFD? | ?U+30000 - U+3FFFD?
   | ?U+40000 - U+4FFFD? | ?U+50000 - U+5FFFD? | ?U+60000 - U+6FFFD?
   | ?U+70000 - U+7FFFD? | ?U+80000 - U+8FFFD? | ?U+90000 - U+9FFFD?
   | ?U+A0000 - U+AFFFD? | ?U+B0000 - U+BFFFD? | ?U+C0000 - U+CFFFD?
   | ?U+D0000 - U+DFFFD? | ?U+E0000 - U+EFFFD? | ?U+F0000 - U+FFFFD?
   | ?U+100000 - U+10FFFD? ;
```

Although it is in the CIF 2.0 character set, the zero-width non-breaking space character, ***U+FEFF***, is not matched by the above productions. It is permitted to appear only as the first character of a CIF 2.0 document, where, if present, it serves to help general-purpose file utilities recognize the file's character encoding. It is not otherwise significant.

2.2.3.2.2 Whitespace and Comments

CIF 2.0 provides for comments introduced by a '#' character and continuing to the end of the line.

```
cif2-comment = '#', { cif2-inline-char } ;
```

A comment must be followed by a line terminator sequence (*cif2-line-term*) or by the end of the CIF. These are not defined as part of the comment, and the formal syntax description ensures that the context provides the line terminator where it is needed. For example, as in CIF 1.1, the trailing terminator might come from the opening delimiter of a text field (§2.2.3.2.4).

Example:

```
# This is a comment, ending at the full point.
```

Although CIF 2.0 is somewhat simplified relative to CIF 1.1 by its context-independent definition of lines of characters, it is still sensitive to line boundaries, therefore simpler productions for some of the constructs to be presented later are made possible by the introduction of several varieties of whitespace runs. First, the usual syntactic separator is a nonempty run of whitespace and possibly comments, neither beginning nor ending with a comment, possibly spanning multiple lines:

```
cif2-wspace = ( cif2-blank-char | cif2-line-term ), cif2-wspace-any ;
```

```
cif2-wspace-any = { cif2-blank-char | ( [ cif2-comment ], cif2-line-term ) } ;
```

The symbol *cif2-wspace* represents what this chapter means by the unqualified term “whitespace” as it applies to CIF 2.0 syntax. In certain contexts, however, we require a similar run of whitespace and comments that ends with the end of a line, or alternatively, one that does not span lines:

```
cif2-wspace-lines = [ cif2-inline-wspace, [ cif2-comment ] ], cif2-line-term,
   { cif2-wspace-to-eol } ;
```

```
cif2-wspace-to-eol = { cif2-blank-char }, [ cif2-comment ], cif2-line-term ;
```

```
cif2-inline-wspace = cif2-blank-char, { cif2-blank-char } ;
```

2.2.3.2.3 Reserved character sequences

CIF 2.0 has the same reserved character sequences as CIF 1.1 (`data_` , `loop_` , `save_`, `global_`, and `stop_`; case-insensitive):

```
cif2-data-token = ( 'D' | 'd' ), ( 'A' | 'a' ), ( 'T' | 't' ), ( 'A' | 'a' ),
   '_';

cif2-save-token = ( 'S' | 's' ), ( 'A' | 'a' ), ( 'V' | 'v' ), ( 'E' | 'e' ),
'_';

cif2-loop-token = ( 'L' | 'l' ), ( 'O' | 'o' ),( 'O' | 'o' ), ( 'P' | 'p' ) ,
   '_' ;

cif2-global-token = ( 'G' | 'g' ), ( 'L' | 'l' ), ( 'O' | 'o' ), ( 'B' | 'b' ),
   ( 'A' | 'a' ), ( 'L' | 'l' ), '_' ;

cif2-stop-token = ( 'S' | 's' ), ( 'T' | 't' ), ( 'O' | 'o' ), ( 'P' | 'p' ),
   '_' ;
```

Also as in CIF 1.1, whitespace-delimited ("unquoted") data values beginning with a dollar sign ($) are reserved. Unquoted data values beginning with the opening or closing square bracket remain forbidden, and indeed, as described in the next section, neither those characters nor the opening and closing braces ({}) can appear anywhere in CIF 2.0 unquoted data values. This prevents any syntactic ambiguity with respect to list and table data values.

2.2.3.2.4 Simple data values

CIF 2.0 simple data values represent sequences of characters without any internal structure significant for CIF's purposes.[2] They are presented in one of four forms. They may be whitespace-delimited strings, whose allowed form varies depending on whether they appear at the start of a line:

```
cif2-wsdelim-string = cif2-wsdelim-string-sol | ( ';',
   { cif2-non-lead-char } ) ;

cif2-wsdelim-string-sol = ( cif2-ordinary-char, { cif2-non-lead-char } )
   - ( ( ( cif2-data-token | cif2-save-token ), { cif2-non-blank-char } )
         | cif2-loop-token | cif2-global-token | cif2-stop-token ) ;

cif2-ordinary-char = cif2-non-lead-char - ( '"' | '#' | '$' | "'" | ';' | '_' )
   ;

cif2-non-lead-char = cif2-non-blank-char - ( '[' | ']' | '{' | '}' ) ;
```

Examples:

```
Lauë
12.345(6)
```

They may be strings contained within a single line and delimited by single quotation marks or single apostrophes:

2 The previous edition of this volume used the term "simple data value" differently, to describe character sequences that satisfy the criteria for CIF 1.1 whitespace-delimited data values.

```
cif2-single-quoted-string =
   ( cif2-quote-delim, cif2-quote-content, cif2-quote-delim )
   | ( cif2-apos-delim, cif2-apos-content, cif2-apos-delim ) ;

cif2-quote-content = { cif2-inline-char - cif2-quote-delim } ;

cif2-quote-delim = '"' ;

cif2-apos-content = { cif2-inline-char - cif2-apos-delim } ;

cif2-apos-delim = "'" ;
```

Examples:

```
'reciprocal lattice'
"O'Malley & Smith"
```

They may be strings, possibly spanning two or more lines, delimited by treble quotation marks or by treble apostrophes:

```
cif2-triple-quoted-string =
   ( cif2-quote3-delim, cif2-quote3-content, cif2-quote3-delim )
   | ( cif2-apos3-delim, cif2-apos3-content, cif2-apos3-delim ) ;

cif2-quote3-delim = '"""' ;

cif2-quote3-content = { [ '"', [ '"' ] ], cif2-non-quote,
   { cif2-non-quote } } ;

cif2-non-quote = cif2-char - '"' ;

cif2-apos3-delim = "'''" ;

cif2-apos3-content = { [ "'", [ "'" ] ], cif2-non-apos, { cif2-non-apos } } ;

cif2-non-apos = cif2-char - "'" ;
```

Examples:

```
"""Multiline
text is supported"""
'''O'Malley & Krüger'''
```

They may be text fields, with the same form as CIF 1.1 text fields:

```
cif2-text-field = cif2-text-delim, cif2-text-content, cif2-text-delim ;

cif2-text-content = { cif2-char } - ( { cif2-char }, cif2-text-delim,
   { cif2-char } ) ;

cif2-text-delim = cif2-line-term, ';' ;
```

Example:

```
;Multiple
lines
;
```

Because they have similar significance and syntactic requirements, it is convenient to refer to single- and triple-quoted strings together simply as quoted strings.

```
cif2-quoted-string = cif2-single-quoted-string | cif2-triple-quoted-string ;
```

Under no circumstance may any of these forms of data value contain their delimiter within. The capability of CIF 1.1 apostrophe- or quotation-mark-delimited string data values to do so is not carried forward into CIF 2.0.

Like CIF 1.1 data values, whether a CIF 2.0 simple data value is presented in delimited or undelimited form is significant in CIF 2.0, but the variety of delimiter is not, except in the sense of whether the syntax affords a representation in each form at all. Furthermore, whatever the form in which it is presented, the content of a data value is case sensitive. Except for converting between line termination styles, changing any character to a different one, no matter how the two are related, produces a distinct value from a general CIF perspective.

In order to enable CIF 2.0 to represent arbitrary values, text field contents as represented by the *cif2-text-content* production are required to be further interpreted in light of the text prefixing protocol described in section (§2.2.4.2.2.1) and the line-folding protocol described in section (§2.2.4.2.2.2).

Text fields have implicit leading whitespace because their delimiting semicolons must appear as the first character of a line. All other data values must be separated from any preceding data name or data value by whitespace, or from a preceding comment by at least an end-of-line:

```
cif2-wspace-data-value = ( cif2-wspace, cif2-nospace-value )
   | ( [ cif2-wspace-lines ], cif2-inline-wspace, cif2-wsdelim-string )
   | ( cif2-wspace-lines, cif2-wsdelim-string-sol )
   | ( [ cif2-wspace, [ cif2-comment ] ], cif2-text-field ) ;

cif2-nospace-value = cif2-quoted-string | cif2-list | cif2-table ;
```

The *cif2-list* and *cif2-table* symbols in the above production are defined in the following section.

2.2.3.2.5 Compound data values

CIF 2.0 provides representations for two kinds of compound values:

- the **list**, which is an ordered sequence of zero or more data values, and
- the **table**, which is an unordered set of zero or more entries binding string keys to data values of any type.

The data values contained within values of these types may be of any type or types, including these compound types themselves. These specifications impose no limit on nesting depth, nor any requirement that the values contained in the same compound value be homogeneous in type.

Syntactically, a list value takes the form of a whitespace-separated sequence of CIF 2.0 data values,

enclosed in square brackets:

```
cif2-list = '[', [ cif2-list-values-start, { cif2-wspace-data-value } ],
   [ cif2-wspace ], ']' ;

cif2-list-values-start = ( cif2-wspace-any, cif2-nospace-value )
   | ( cif2-wspace-any, [ cif2-comment ], cif2-text-field )
   | ( [ { cif2-wspace-to-eol }, cif2-inline-wspace ], cif2-wsdelim-string )
   | ( cif2-wspace-to-eol, { cif2-wspace-to-eol }, cif2-wsdelim-string-sol ) ;
```

There is no requirement for whitespace between the delimiting brackets of a list and the values within, for whitespace between the brackets of an empty list, or for any spaces or tabs between the opening bracket of a list and an initial comment syntactically contained within, but arbitrary whitespace is permitted at any or all of these locations.

Examples:

```
[ 0.0 0.5 0.0 ]

[]
```

A table value takes the form a whitespace separated series of zero or more entries, each consisting of a label formed by appending a colon (:) to a CIF 2.0 single- or triple-quoted string, and a value of any CIF 2.0 data value type:

```
cif2-table = '{', [ cif2-wspace-any, cif2-table-entry,
   { cif2-wspace, cif2-table-entry } ], [ cif2-wspace ], '}' ;

cif2-table-entry = cif2-quoted-string, ':',
   ( cif2-nospace-value | cif2-wsdelim-string | cif2-wspace-data-value ) ;
```

Whitespace is not required between the label's terminating colon and the associated data value. Likewise, there is no requirement for whitespace between the delimiting braces of a table and the entries within, for whitespace between the braces of an empty table, or for any spaces or tabs between the opening brace of a table and an initial comment syntactically contained within. Arbitrary whitespace is nevertheless permitted at all those locations.

Examples:

```
{ 'A': alanine
  'R': arginine
  'N': asparagine
  'D': 'aspartic acid' }

{
  """red""":  [ 1.0 0.0 0.0 ] "yellow": [ 1.0 1.0 0.0 ] 'green': [ 0.0 1.0 0.0 ]
  '''cyan''':[ 0.0 1.0 1.0 ] "blue":  [ 0.0 0.0 1.0 ] "magenta":[ 1.0 0.0 1.0 ]
}

{}
```

2.2.3.2.6 Data names

CIF data names begin with an underscore character and may contain any non-whitespace character:

```
cif2-data-name = '_' , cif2-non-blank-char, { cif2-non-blank-char } ;
```

CIF 2.0 does not directly restrict data name lengths, but data names cannot span lines, and lines, at least in the native syntax presented here, are limited to 2048 characters. Data names are case-insensitive; transforming one by changing the alphabetic case of one or more of its letters neither changes its significance nor is relevant to data blocks' and save frames' data-name uniqueness requirements.

Examples:

```
_DDL1_style
_DDL2.style
_DDLm.style
```

2.2.3.2.7 Data items

CIF 2.0 data values, whether simple or compound, can be associated with data names to form data items in the same ways that such associations are formed in CIF 1.1. Where compound values appear, it is only the outermost data values that are bound to data names – those that do not appear as part of other values. One value can be paired with one name to form one data item by simple juxtaposition. Alternatively, one or more collections of one or more correlated data values can be be associated with one set of data names in a loop construct to form multiple items, one for each value:

```
cif2-data = cif2-scalar-item | cif2-loop ;

cif2-scalar-item = cif2-data-name, cif2-wspace-data-value ;

cif2-loop = cif2-loop-token, cif2-wspace,
   cif2-data-name, { cif2-wspace, cif2-data-name },
   cif2-wspace-data-value, { cif2-wspace-data-value } ;
```

As in CIF 1.1, the number of data names in a loop must evenly divide the number of values in it, and only the sequence of data values, not any aspect of the amount or type of whitespace between, that is significant for associating values with names and with each other.

Examples:

```
_name.category_id  units

_import.get        [{'save':units_code  'file':templ_enum.cif}]

 loop_
    _dictionary_valid.application
    _dictionary_valid.attributes
  [Item  Mandatory]      ['_definition.id'  '_definition.update'
                          '_name.object_id'  '_name.category_id'
                          '_type.container'  '_type.contents'
  [Item  Recommended]    ['_definition.scope'  '_definition.class'
                          '_type.source'  '_type.purpose'
                          '_description.text'  '_description.common']
```

```
[Item  Prohibited]     [CATEGORY  DICTIONARY]
```

2.2.3.2.8 Data blocks and save frames

CIF 2.0 data items are contained within data blocks and save frames, just as in CIF 1.1, with the same requirements for data names' uniqueness within their containers and frame codes' uniqueness within their data blocks. The beginning of a data block is marked by a data block heading consisting of the concatenation of a five-character sequence case-insensitively matching "data_" with the one or more non-whitespace characters of that block's block code. The end of a data block is implicit at the beginning of the next data block or the end of the file, whichever comes first. Data blocks do not nest.

Data blocks may contain save frames, interspersed with data in any sequence. To date, no use of save frames is envisioned for data files; these remain limited to dictionary files in practice. Other than the larger character set with which their frame codes can be formed, save frames have the same structure in CIF 2.0 as in CIF 1.1.

```
cif2-data-block = cif2-data-heading, { cif2-block-content } ;

cif2-data-heading = cif2-data-token, cif2-container-code ;

cif2-block-content = cif2-wspace, ( cif2-data | cif2-save-frame ) ;

cif2-save-frame = cif2-save-heading, { cif2-frame-content }, cif2-wspace,
   cif2-save-token ;

cif2-save-heading = cif2-save-token, cif2-container-code ;

cif2-frame-content = cif2-wspace, cif2-data ;

cif2-container-code = cif2-non-blank-char, { cif2-non-blank-char } ;
```

Unlike CIF 1.1, CIF 2.0 places no direct constraint on the lengths of block codes and frame codes, but these are indirectly constrained by the overall 2048-character line length limit.

Examples:

```
data_CIF_CORE

_dictionary.title                       CIF_CORE
_dictionary.version                     3.0.04
_dictionary.date                        2015-02-18

_description.text
;
    The CIF_CORE dictionary records all the CORE data items defined
    and used with in the Crystallographic Information Framework (CIF).
;

save_CIF_CORE

_definition.id                          CIF_CORE
_definition.scope                       Category
_definition.class                       Head
_definition.update                      2014-06-18
```

```
_description.text
;
     The CIF_CORE group contains the definitions of data items that
     are common to all domains of crystallographic studies.
;
_name.category_id                       CIF_DIC
_name.object_id                         CIF_CORE

save_
```

2.2.3.2.9 CIF 2.0 file

An overall CIF 2.0 document begins with a leading code identifying the syntax version, which has a form that could also be interpreted as an ordinary comment:

```
#\#CIF_2.0
```

Additional leading whitespace is permitted after the version code, followed by a sequence of zero or more data blocks. All data blocks in the same CIF must have distinct block codes, as evaluated in the case-insensitive sense previously discussed. CIF 2.0 places no minimum requirement on data content; only the version-identification comment is required.

```
cif2-cif = ( cif2-file-heading,
   [ cif2-line-term, cif2-wspace-any, [ cif2-data-blocks | cif2-comment ] ] )
   - ( { cif2-char }, 2049 * cif2-inline-char, { cif2-char } );

cif2-file-heading = [ ?U+FEFF? ], cif2-version-code, [ cif2-inline-wspace ] ;

cif2-version-code = '#\#CIF_2.0' ;

cif2-data-blocks = cif2-data-block, { cif2-wspace, cif2-data-block },
   [ cif2-wspace, [ cif2-comment ] ] ;
```

The line-length restriction is expressed once for all on the overall *cif2-cif* production. Specifically, sequences of 2049 or more characters that fail to span at least two lines are excluded; as in CIF 1.1, line terminators do not count toward the 2048-character limit. Figure 3 presents a simple, syntacitically complete CIF expressed using CIF 2.0 syntax.

2.2.3.2.10 Comparison with STAR 2.0

CIF 2.0 native syntax is a derived from STAR 2.0 syntax, but it is not strictly a subset. There are two incompatibilities between the two:

- In STAR 2.0, list elements and table entries are separated from each other by commas (and optional whitespace) whereas in CIF 2.0 these elements are separated by mandatory whitespace alone.
- STAR 2.0 requires files to contain at least one data block, whereas STAR 1.0 and all versions of CIF to date permit files to contain no data blocks at all.

Additionally, CIF 2.0 places several restrictions relative to STAR 2.0:

- Save frames may not be nested in CIF.
- CIF does not permit nested loops.
- CIF documents may not contain `global_` sections.
- CIF requires that data names, data block codes, and save frame codes be unique within their scopes in a case-insensitive sense, whereas STAR requires only exact uniqueness.
- CIF does not recognize STAR 2.0's mechanism for embedding string delimiters, nor is the escape character on which it is based, ***U+0007***, in CIF 2.0's allowed set.
- CIF does not recognize STAR save frame references, and it reserves whitespace-delimited values having the form of STAR frame references.
- CIF does not recognize or allow STAR 2.0 ref-tables.
- CIF 2.0 does not allow whitespace in table keys between the quoted string and its immediately following colon.
- The character set supported by CIF 2.0 is a slight restriction of the one supported by STAR 2.0.
- CIF imposes a 2048-character limit on line lengths.
- CIF 2.0 requires files to start with a CIF version comment.

A CIF document can be converted to STAR 2.0 syntax by inserting comma separators into any list or table data values, and by inserting an empty data block if otherwise no data blocks are present. A conforming STAR 2.0 document that satisfies CIF 2.0's restrictions can be converted to CIF 2.0 syntax by changing the comma delimiters within its list and table data values to space characters. Such conversions can readily be automated.

2.2.3.2.11 Complete formal grammar

The CIF 2.0 formal syntax and grammar were presented piecemeal in preceding sections. This section presents the full set of productions in conventional top-down order.

```
cif2-cif = ( cif2-file-heading,
   [ cif2-line-term, cif2-wspace-any, [ cif2-data-blocks | cif2-comment ] ] )
   - ( { cif2-char }, 2049 * cif2-inline-char, { cif2-char } );

cif2-file-heading = [ ?U+FEFF? ], cif2-version-code, [ cif2-inline-wspace ] ;

cif2-version-code = '#\#CIF_2.0' ;

cif2-data-blocks = cif2-data-block, { cif2-wspace, cif2-data-block },
   [ cif2-wspace, [ cif2-comment ] ] ;

cif2-data-block = cif2-data-heading, { cif2-block-content } ;

cif2-data-heading = cif2-data-token, cif2-container-code ;
```

```
cif2-block-content = cif2-wspace, ( cif2-data | cif2-save-frame ) ;

cif2-save-frame = cif2-save-heading, { cif2-frame-content }, cif2-wspace,
   cif2-save-token ;

cif2-save-heading = cif2-save-token, cif2-container-code ;

cif2-frame-content = cif2-wspace, cif2-data ;

cif2-container-code = cif2-non-blank-char, { cif2-non-blank-char } ;

cif2-data = cif2-scalar-item | cif2-loop ;

cif2-scalar-item = cif2-data-name, cif2-wspace-data-value ;

cif2-loop = cif2-loop-token, cif2-wspace,
   cif2-data-name, { cif2-wspace, cif2-data-name },
   cif2-wspace-data-value, { cif2-wspace-data-value } ;

cif2-data-name = '_' , cif2-non-blank-char, { cif2-non-blank-char } ;

cif2-wspace-data-value = ( cif2-wspace, cif2-nospace-value )
   | ( [ cif2-wspace-lines ], cif2-inline-wspace, cif2-wsdelim-string )
   | ( cif2-wspace-lines, cif2-wsdelim-string-sol )
   | ( [ cif2-wspace, [ cif2-comment ] ], cif2-text-field ) ;

cif2-nospace-value = cif2-quoted-string | cif2-list | cif2-table ;

cif2-list = '[', [ cif2-list-values-start, { cif2-wspace-data-value } ],
   [ cif2-wspace ], ']' ;

cif2-list-values-start = ( cif2-wspace-any, cif2-nospace-value )
   | ( cif2-wspace-any, [ cif2-comment ], cif2-text-field )
   | ( [ { cif2-wspace-to-eol }, cif2-inline-wspace ], cif2-wsdelim-string )
   | ( cif2-wspace-to-eol, { cif2-wspace-to-eol }, cif2-wsdelim-string-sol ) ;

cif2-table = '{', [ cif2-wspace-any, cif2-table-entry,
   { cif2-wspace, cif2-table-entry } ], [ cif2-wspace ], '}' ;

cif2-table-entry = cif2-quoted-string, ':',
   ( cif2-nospace-value | cif2-wsdelim-string | cif2-wspace-data-value ) ;

cif2-quoted-string = cif2-single-quoted-string | cif2-triple-quoted-string ;

cif2-wsdelim-string = cif2-wsdelim-string-sol
   | ( ';', { cif2-non-lead-char } ) ;

cif2-wsdelim-string-sol = ( cif2-ordinary-char, { cif2-non-lead-char } )
   - ( ( ( cif2-data-token | cif2-save-token ), { cif2-non-blank-char } )
         | cif2-loop-token | cif2-global-token | cif2-stop-token ) ;

cif2-ordinary-char = cif2-non-lead-char - ( '"' | '#' | '$' | "'" | ';' | '_' )
   ;

cif2-non-lead-char = cif2-non-blank-char - ( '[' | ']' | '{' | '}' ) ;
```

```
cif2-single-quoted-string =
   ( cif2-quote-delim, cif2-quote-content, cif2-quote-delim )
   | ( cif2-apos-delim, cif2-apos-content, cif2-apos-delim ) ;

cif2-quote-content = { cif2-inline-char - cif2-quote-delim } ;

cif2-quote-delim = '"' ;

cif2-apos-content = { cif2-inline-char - cif2-apos-delim } ;

cif2-apos-delim = "'" ;

cif2-triple-quoted-string =
   ( cif2-quote3-delim, cif2-quote3-content, cif2-quote3-delim )
   | ( cif2-apos3-delim, cif2-apos3-content, cif2-apos3-delim ) ;

cif2-quote3-delim = '"""' ;

cif2-quote3-content = { [ '"', [ '"' ] ], cif2-non-quote,
   { cif2-non-quote } } ;

cif2-non-quote = cif2-char - '"' ;

cif2-apos3-delim = "'''" ;

cif2-apos3-content = { [ "'", [ "'" ] ], cif2-non-apos, { cif2-non-apos } } ;

cif2-non-apos = cif2-char - "'" ;

cif2-text-field = cif2-text-delim, cif2-text-content, cif2-text-delim ;

cif2-text-content = { cif2-char } - ( { cif2-char }, cif2-text-delim,
   { cif2-char } ) ;

cif2-text-delim = cif2-line-term, ';' ;

cif2-data-token = ( 'D' | 'd' ), ( 'A' | 'a' ), ( 'T' | 't' ), ( 'A' | 'a' ),
   '_';

cif2-save-token = ( 'S' | 's' ), ( 'A' | 'a' ), ( 'V' | 'v' ), ( 'E' | 'e' ),
'_';

cif2-loop-token = ( 'L' | 'l' ), ( 'O' | 'o' ),( 'O' | 'o' ), ( 'P' | 'p' ) ,
   '_' ;

cif2-global-token = ( 'G' | 'g' ), ( 'L' | 'l' ), ( 'O' | 'o' ), ( 'B' | 'b' ),
   ( 'A' | 'a' ), ( 'L' | 'l' ), '_' ;

cif2-stop-token = ( 'S' | 's' ), ( 'T' | 't' ), ( 'O' | 'o' ), ( 'P' | 'p' ),
   '_' ;

cif2-comment = '#', { cif2-inline-char } ;

cif2-wspace = ( cif2-blank-char | cif2-line-term ), cif2-wspace-any ;

cif2-wspace-any = { cif2-blank-char | ( [ cif2-comment ], cif2-line-term ) } ;

cif2-wspace-lines = [ cif2-inline-wspace, [ cif2-comment ] ], cif2-line-term,
   { cif2-wspace-to-eol } ;
```

```
cif2-wspace-to-eol = { cif2-blank-char }, [ cif2-comment ], cif2-line-term ;

cif2-inline-wspace = cif2-blank-char, { cif2-blank-char } ;

cif2-char = cif2-inline-char | ?U+000A? | ?U+000D? ;

cif2-line-term = ( ?U+000D?, [ ?U+000A? ] ) | ?U+000A? ;

cif2-inline-char = cif2-non-blank-char | cif2-blank-char ;

cif2-blank-char = ?U+0020? | ?U+0009? ;

cif2-non-blank-char = ?U+0021 - U+007E? | ?U+00A0 – U+D7FF?
    | ?U+E000 - U+FDCF?   | ?U+FDF0 - U+FEFE?   | ?U+FF00 - U+FFFD?
    | ?U+10000 - U+1FFFD? | ?U+20000 - U+2FFFD? | ?U+30000 - U+3FFFD?
    | ?U+40000 - U+4FFFD? | ?U+50000 - U+5FFFD? | ?U+60000 - U+6FFFD?
    | ?U+70000 - U+7FFFD? | ?U+80000 - U+8FFFD? | ?U+90000 - U+9FFFD?
    | ?U+A0000 - U+AFFFD? | ?U+B0000 - U+BFFFD? | ?U+C0000 - U+CFFFD?
    | ?U+D0000 - U+DFFFD? | ?U+E0000 - U+EFFFD? | ?U+F0000 - U+FFFFD?
    | ?U+100000 - U+10FFFD? ;
```

2.2.3.3 Differences between CIF 1.1 and CIF 2.0

Although CIF 1.1 syntax and CIF 2.0 syntax have similar form, files written in CIF 1.1 syntax rarely conform to CIF 2.0 – at minimum, they are unlikely to bear a CIF-version comment designating CIF 2.0 – and files written in CIF 2.0 syntax typically do not conform to CIF 1.1. Moreover, CIF 2.0 files that do conform syntactically to CIF 1.1 may be interpreted differently when processed as CIF 1.1 than they are when processed as CIF 2.0. In short, CIF 2.0 is not backwards compatible with CIF 1.1. The differences between the two versions are summarized in the following sections.

2.2.3.3.1 Version code

The content of every conforming CIF 2.0 file begins with a structured comment identifying the file as conforming to CIF 2.0. The content of a CIF 1.1 file optionally begins with a similar comment identifying its format. These CIF-version comments are intended to help software determine how to interpret the contents. A file without such a comment or with a comment specifying version 1.1 fails to conform to CIF 2.0, even if only for that reason, but such a file may conform to CIF 1.1. On the other hand, CIF 1.1 does not require a version comment be present, and if one is present then CIF 1.1 does not constrain the version number it designates. Therefore, a document that bears a version comment designating CIF 2.0 does not necessarily fail to conform to CIF 1.1.

Syntactically, the version-comment line of a CIF 2.0 document must not contain any non-whitespace characters after the version code, whereas CIF 1.1 is more permissive – it requires only that the version code be followed immediately by whitespace.

2.2.3.3.2 Character set and encoding

Whereas CIF 1.1 native format describes text conforming to local convention and relying only on characters drawn from a 98-character subset of those mapped by ASCII, CIF 2.0 native format is a

binary format resembling Unicode text encoded in UTF-8 and drawing on nearly all Unicode characters. CIF 2.0's reliance specifically on UTF-8 and the fundamentally binary nature following from that do not constitute a distinction in practice on many common systems, however. UTF-8 is a common choice of character encoding for computer systems and electronic documents, and it encodes the characters of the CIF 1.1 character set consistently with several other common choices, such as the ISO-8859 family of character sets and Windows-1252. As a result, many (locally) conforming CIF 1.1 files already conform in fact with CIF 2.0's character set and encoding requirements.

2.2.3.3.3 White space and line termination

Whereas CIF 1.1 native format relies on local conventions for text files to define the division of instance documents into lines of text, CIF 2.0 defines division into lines according to a specific set of line-terminating character sequences. In practice, however, the line termination sequences recognized by CIF 2.0 are those employed by the text conventions of those major computer operating systems of CIF's lifetime whose text conventions rely on line-terminating character sequences. A CIF 1.1 text file conforming to such conventions will also conform to CIF 2.0 line termination conventions, subject only to a possible transcoding to UTF-8.

Additionally, however, characters and character sequences that CIF 2.0 always recognizes as line terminators will be treated instead as inline whitespace under CIF 1.1 where they are not recognized as line terminators according to applicable local convention. This can change the meaning of a CIF conforming to one syntax if it is reinterpreted according to the other without conversion, but it is of no consequence for files, of either variety, that use the carriage return and line feed characters only according to local line termination convention.

2.2.3.3.4 Identifiers

All CIF 1.1 data names, block codes, and frame codes are valid in CIF 2.0, but CIF 2.0 allows identifiers that CIF 1.1 does not, because (i) CIF 2.0 identifiers can draw characters from substantially all of Unicode, and (ii) CIF 2.0 does not place any limit on identifier lengths other than the overall 2048-character line-length limit. Additionally, although the definition of identifier uniqueness in §2.2.2 applies equally to CIF 1.1 and CIF 2.0, software predating the advent of CIF 2.0 typically implements a simplified variation that works only for CIF 1.1's more limited character set.

2.2.3.3.5 Data values

The two CIF versions exhibit several differences with respect to the representation of data values. Most notably, CIF 2.0 adds triple-quoted, multiline strings, lists, and tables, but it also differs from CIF 1.1 with respect to each of the data value forms the two share.

In addition to the expanded range of whitespace-delimited data values afforded by CIF 2.0's larger character set, CIF 2.0 differs from CIF 1.1 in that it disallows the opening and closing square bracket and opening and closing brace characters ([,], {, }) appearing anywhere in such a data value. CIF 1.1 forbids the former two from appearing as the first character of an unquoted data value, but does not

otherwise restrict their appearance in such values. The new CIF 2.0 restrictions avoid ambiguities that otherwise could arise with its list and table data types.

Furthermore, CIF 2.0, unlike CIF 1.1, does not permit apostrophe-delimited or quote-delimited data values to embed their delimiter under any circumstance. On one hand, this means that some valid CIF 1.1 lexical values (e.g. `'O'Sullivan'`) are syntactically invalid in CIF 2.0. Generally, the same content can be conveyed in triple-quoted form in CIF 2.0, and it can always be conveyed in text-field form. On the other hand, this means that CIF 2.0 triple-quoted data values that do not span lines (*e.g.* `'''O'Sullivan'''`) may conform to the CIF 1.1 syntax for (single-) quoted values, but they will not be interpreted the same way in CIF 1.1 as they would be in CIF 2.0. Such values can often be converted to an equivalent single-quoted CIF 1.1 form, and always to an equivalent text-field form.

Finally, CIF 2.0 requires that text field contents be interpreted according to the text-prefixing and line-folding protocols where the form of those contents so indicates (§§2.2.4.2.2.1-2.2.4.2.2.2), but this is optional in CIF 1.1. In practice, CIF 1.1 processors vary: many, but not all, processors recognize and implement the line-folding protocol for text fields, but only a few also implement the text-prefixing protocol.

2.2.4 CIF Semantics

The scientific meaning associated with objects in the CIF syntactical data model described above is supplied by the CIF dictionaries, as described in Section x. In order to create the link to the dictionary information, a syntax forming part of the CIF framework must specify how the objects in the syntactical data model align with the data model and types defined in those CIF dictionaries (for more detail see Section x.x.x.x). Table xxxx provides an overview of the correspondence between objects in the CIF data model and the DDL dictionary data model, and the following sections discuss these alignments in more detail.

Note that the particular dictionary or dictionaries to which a given CIF data block conforms are indicated using the `_audit.conform` data name (see p xxx). It should be emphasized that CIF processing software is not expected to interpret the textual definitions in the indicated dictionaries; rather, the human programmer interprets definitions when the software is constructed. Therefore, the most common use for the `_audit.conform` tag is to check that the dictionary upon which the programmer has based their CIF- reading software corresponds to the dictionary used by the programmer of the CIF writing software to construct the data block.

It is inevitable that CIF reading software will encounter data names that the programmer was not aware of when constructing the software. Such a situation might arise due to the presence of local data names (see x.x.x.x), use of a tag reserved for use by a particular group or organisation (see x.x.x.x), or admixture of tags from dictionaries that the reading software is not prepared for. This situation is not erroneous. In such cases, correct interpretation and manipulation of the data values for those data names must usually rely on context and sources of information external to the CIF framework. In

addition, the CIF syntax allows some simple semantics to be deduced. Section x.x.x.x below describes those semantics that can safely be assumed when working with CIF data names for which no dictionary definition is available.

The transformations from the contents of the EBNF productions to the abstract dictionary types described below can apply to any format that yields a sequence of characters for objects in the data model. It is recommended that alternative syntaxes for encapsulating CIF data preserve these transformations as the default.

2.2.4.1 Data names

Text matched to the production `cif1-data-name` or `cif2-data-name` corresponds directly to a data name in the CIF data model. These link data values with definitions in a DDL dictionary, and thereby specify values' interpretations. A single special case applies when a standard uncertainty is appended to the data value (see 2.4.2.x) associated with a given data name. In this case, some CIF dictionary formalisms associate the standard uncertainty value with a separate data name formed by appending the characters "_su" to the base data name (see 2.x.x.x), as if the value and its uncertainty were specified as separate data items.

2.2.4.2 Data values

The CIF data model requires that every data value be associated with a data name. The CIF syntax creates this association either by presenting the value as part of a data name / data value pair or by using a loop structure (see section 2.2.2). The latter approach assigns one or more sets of values to a single set of data names, and it is used primarily when dealing with data names whose dictionary definitions afford multiple distinct values in the same data set, especially where multiple values associated with different data names must be grouped together into packets for proper interpretation. In a loop containing `N` data names, data values are associated with the same packet when they have the same value for the integer part of `S/N`, where `S` is the value's ordinal position among all the values of the loop, numbered from 0. Each data value is associated with the data name whose ordinal position among the data names of the loop is S reduced modulo N, where the data names are numbered from 0. These data name / data value associations form the items belonging to loop packets. Informally, if a loop is laid out as a table with the data names as column headers then the packets correspond to rows in the table and the data name associated with each value is the header of its column.

These two different mechanisms for forming data items do not carry any inherent semantic difference. That is, given a logical set of data conforming to one data block or save frame in the CIF data model, and a set of data names that each have exactly one data value associated with them in that data set, those data can be expressed equivalently in either syntax via either a single-packet loop construct or separate data name / data value pairs. Both representations convey the same information. Nevertheless, historically, some items have conventionally been presented only via loop constructs and others conventionally only as individual items, and in practice, some software will reject files that

break from those conventions. This is to some extent supported by CIF dictionaries written in the DDL1 dictionary definition language (section x.x.x), which provides a mechanism by which item definitions can specify whether items must be syntactically presented in a loop. The DDL2 (section y.y.y) and DDLm (section z.z.z) languages do not provide for making that artificial distinction.

When reading a CIF file, the character sequences matched by *cif1-wspace-data-value* and *cif2-wspace-data-value* productions are mapped to data values in the CIF data model via a transformation that depends on the alternative by which that production was matched, the syntax version, and details of the value's lexical representation. Three syntactic data types are defined for simple data values: null, unquoted character sequence, and quoted character sequence. Two syntactic types are defined for compound values: list and table. Lexical CIF text is converted to simple data values as described in Table 2.

Table 2: Mapping from CIF syntax to data values

Character content from	**Transformation**	**Syntactic Type**	**Notes**
cif1-wsdelim-string, *cif2-wsdelim-string*	none	null	When the character content is a single query mark (`?`) or full point (`.`)
cif1-wsdelim-string, *cif2-wsdelim-string*	none	unquoted	When the character content is not a single query mark (?) or full point (.)
cif1-quote-content, *cif1-apos-content*, *cif2-quote-content*, *cif2-apos-content*	none	quoted	
cif2-quote3-content, *cif2-apos3-content*	Line terminator normalization (required)	quoted	See 2.2.4.2.2
cif1-text-content	prefix removal (optional); line unfolding (recommended); line terminator normalization (optional)	quoted	See 2.2.4.2.2
cif2-text-content	prefix removal (mandatory); line unfolding (mandatory); line terminator normalization (required)	quoted	See 2.2.4.2.2

Additionally, each character sequence matched by the *cif2-list* production is converted to a list data value constituting an ordered sequence of data values obtained by converting the subsequences matched by the *cif2-list-values-start* and *cif2-wspace-data-value* productions according to the specifications in this section. Similarly, each character sequence matched by the *cif2-table*

production is converted to a table data value constituting an unordered mapping from keys of “quoted” syntactic type to values of any syntactic type. Each maximal subsequence matched by the *cif2-table-entry* production is converted to one such key / value pair, wherein the subsequence within matched by the *cif2-quoted-string* production, converted as for a simple data value, serves as the key, and the one matched by the *cif2-nospace-value*, *cif2-wsdelim-string*, or *cif2-wspace-data-value* production, converted according to the specifications in this section, serves as the value.

The significance of each data item is determined by a definition of its data name. Applications interpret the character sequence and syntactic type of each value in light of that definition, which associates a semantic data type with it and may involve further transformation of the syntactic character sequence. Details of these type mappings are variously DDL-, dictionary-, and application-specific, but some common conventions applied to a wide variety of items are discussed in the following subsections.

2.2.4.2.1 Null values

Null values are special values that formalize the absence of a specific value, and therefore are possible values for all dictionary data types. Null values are represented in CIF 1.1 and CIF 2.0 native syntax are those to which syntactic type “null” is assigned: the query (?) and full point (.) characters presented as unquoted data values. These convey two different senses of a placeholder for an omitted value. They may be associated with any data name.

The query mark presented as a whitespace-delimited data value is an explicit signal that no value is presented, without speaking to the question of whether any value exists that could be appropriately associated with the item's data name in its data set, and if so, what value that may be. A data item with such a value is equivalent to no data item at all, except inasmuch as it may play a place-holding role within loop structures. Such a value may serve a similar place-holding function inside list and table data values as well.

The full-point character presented as a whitespace-delimited data value serves two different, but related, purposes: it can mean that the data item does not apply to the data set, or it can mean that the item takes a default value determined by its definition. As an example of the first use, in a loop presenting both atomic coordinates and anisotropic thermal parameters, the full-point might be used for the anisotropic thermal parameter values of atoms that were refined isotropically. As an example of the second use, the CIF core dictionary defines that the data name `_geom_bond_site_symmetry_1` has a default value of `1_555`, which represents the identity operation. Where that default value indeed applies, CIF bond distance tables expressed using the core dictionary commonly present the unquoted full-point as this item's value. In that use, however, the default value conveys the sense of “no additional symmetry”, which is not well distinguished from “not applicable”. More generally, the second sense of the value can be construed as “not varying from the default”, which differs from “inapplicable” only in the details of terms in which the definition is couched.

2.2.4.2.2 Character types

As described so far, both syntactic forms of CIF defined in this chapter have limitations on the values they can express. Their line-length restrictions impose some of those limitations, but so also do the defined forms of data values – in particular, neither syntax by itself supports embedding general CIF in a CIF data value. CIF 1.1 is furthermore limited by its character repertoire, which is inadequate for directly representing text in languages other than English, and it presents some practical issues surrounding line termination. Approaches to all of these issues and limitations have been devised and are in common use, as described next.

2.2.4.2.2.1 Prefixed text

The text-prefix protocol addresses the problem of embedding CIF or CIF-like text in a CIF data value by defining a format for text-field contents in which a non-empty character sequence is prefixed to each line in a manner that can be recognized and reversed. This solves the problem of embedding text-field delimiters in a text field because prefixes interposed between the beginning of lines and their logical contents ensure that no semicolon in the field content appears at the start of its physical line. The text prefix protocol can also improve the human readability of text-field contents by increasing a visual distinction between the field content and the containing CIF. In combination with the line-folding protocol described in the next section, this allows text fields to represent arbitrary sequences of allowed characters.

The text prefix protocol is substantially the same for both CIF 1.1 and CIF 2.0, but it differs between them according to their differences in character set and line-termination semantics. In either case, a **prefix** consists of a sequence of one or more characters allowed to appear in a text field, excluding the backslash (\), and neither starting with a semicolon nor spanning lines. In CIF 1.1, that can be formalized as

```
cif1-prefix = ( cif1-prefix-start-char, { cif1-prefix-char } )
   - ( { cif1-char }, ?EOL? { cif1-char } ) ;

cif1-prefix-start-char = cif1-prefix-char - ';' ;

cif1-prefix-char = cif1-char - '\' ;
```

Corresponding CIF 2.0 productions would be

```
cif2-prefix = cif2-prefix-start-char, { cif2-prefix-char } ;

cif2-prefix-start-char = cif2-prefix-char - ';' ;

cif2-prefix-char = cif2-inline-char - '\' ;
```

The specific character sequence of a prefix is not meaningful, and it should be noted that whitespace characters not serving a role in line-termination can appear in a prefix. In CIF 1.1, however, which characters those are is system-dependent. It is therefore recommended that authors of CIF 1.1 documents that employ prefixing avoid this issue by excluding the newline and carriage return characters from prefixes altogether, regardless of their local significance for line termination.

The text prefix protocol **applies** to those text fields whose full contents start with a prefix, followed by either one or two backslashes, followed by zero or more whitespace characters up to the end of the first line, and whose remaining lines all start with the same prefix. In CIF 1.1, that would be a text field whose *cif1-text-content* also matches this production:

```
cif1-prefixed-text = cif1-prefix, '\', [ '\' ], [ cif1-inline-wspace ],
   { ?EOL?, cif1-prefix, cif1-inline-chars } ;
```

In CIF 2.2, it would be one whose *cif2-text-content* also matches

```
cif2-prefixed-text = cif2-prefix, '\', [ '\' ], { cif2-blank-char },
   { cif2-line-term, cif2-prefix, { cif2-inline-char } } ;
```

Both are subject to the additional requirement, not expressible in EBNF notation, that the same prefix appear at the beginning of each line of the text content.

Prefix removal is a process for transforming text field contents. When performed on the contents of a text field to which the text prefix protocol does not apply, it produces a result identical to the original contents. When performed on the contents of a text field to which the text prefix protocol does apply, it produces a result in which the prefix has been removed from each line, and

(1) if the remainder of the first line starts with two backslashes then the first is removed;

(2) otherwise, the whole first line is removed, leaving the field either empty or with one fewer line.

For example, the text prefix protocol applies to these data items:

```
_prefixed_only
;>\
>_embedded_text
>;content
>;
;

_folded_and_prefixed
;...\\
...Non-folded line.
...This logical line was\
... folded across multiple \
...lines.
;
```

Prefix removal performed on the content of the first value yields text that has the form of a CIF data name with a text field value:

```
_embedded_text
;content
;
```

Prefix removal performed on the content of the second value yields text whose first line contains only a backslash; the significance of such text is discussed in the next section:

```
\
Non-folded line.
This logical line was\
 folded across multiple \
lines.
```

Prefix removal is not customarily performed on text fields obtained from CIF 1.1 documents, therefore prefixed text fields should not be written in CIF 1.1 intended to be presented or archived outside environments having a local convention of performing prefix removal. On the other hand, CIF 2.0 syntax carries a requirement to interpret all text fields as if prefix removal were performed on their lexical contents. Thus, although it is presented as a semantic consideration, text prefixing has a strong syntactic character, especially in CIF 2.0 – it is about which character sequences are conveyed by text field values, not about the significance of those values.

2.2.4.2.2.2 Long lines

The line-folding protocol addresses the problem of representing data for which line breaks are significant, and which furthermore contain spans of at least 2049 characters between line breaks (or 2048 before the first line break). Such data cannot be expressed directly in either CIF syntax because of the limitation on line lengths. Line folding involves formatting text-field contents by splitting (folding) some of their logical lines across multiple physical lines, in a manner that can be recognized and reversed. It is substantially the same in CIF 1.1 and CIF 2.0. Where one logical line is split across multiple physical lines according to the protocol, each physical line but the last ends with a backslash character and possibly trailing whitespace. The last physical line of a line-folded text field's contents may optionally end with such a sequence as well.

The line-folding protocol applies to those text fields whose contents begin with a backslash as the first character, with no other non-whitespace characters preceding the end of the first line. Where text prefix processing is also being performed (previous section), it is the results of prefix removal that are subject to this analysis and to the unfolding process described next. Usually, some physical lines of a field to which line folding applies will end with a backslash and optional whitespace, and others will not.

Unfolding is a process for transforming text field contents. When performed on the contents of a text field to which the line-folding protocol does not apply, it produces a result identical to the original contents. When performed on the contents of a text field to which the line-folding protocol does apply, it produces a result in which each logical line occupies exactly one physical line. Specifically, wherever a backslash is the last non-blank character of a line, that backslash and everything following it on the line are removed, and the remainder is joined to the next line, if any.

When preparing folded text field contents, it is necessary to take care to avoid altering the value. In particular, if any logical line contains a backslash as its last non-blank character, then it will need to be folded by adding another backslash at the end and inserting an empty line after. Similarly, the line-folding protocol can be applied to protect significant trailing whitespace on a line of a text field, which

CIF otherwise permits to be ignored (see next section).

For example, the line-folding protocol applies to this data item:

```
_folded
;\
Non-folded line.
This logical line was\
 folded across multiple \
lines.
;
```

Unfolding performed on the value's content yields this text:

```
Non-folded line.
This logical line was folded across multiple lines.
```

Performing unfolding on text fields obtained from CIF 1.1 documents is optional, but recommended. The form of a text field to which the protocol applies is distinctive enough that performing unfolding is less likely to produce a misinterpretation than is not doing so. On the other hand, CIF 2.0 syntax carries a requirement to interpret all text fields as if unfolding were performed on their contents, after prefix removal. Thus, like text prefixing, although line folding is presented as a semantic consideration, it has a strong syntactic character, especially in CIF 2.0 – it is about the details of which character sequences are conveyed by text field values.

Additionally, if one considers the general problem of transforming CIF documents with longer lines to equivalent documents with shorter lines then the question arises of what to do about long comment lines. Other than an initial CIF-version comment, CIF attributes no semantic significance to comments, so in that sense, the meaning of a CIF is unchanged by any transformation of comments to comments or whitespace, provided only that CIF-version comments are neither modified nor moved. Nevertheless, there is a variation of the line-folding protocol that can preserve comment text over such transformations. According to this variation, a comment containing a backslash as the first character of its text and no other non-blank characters before the end of the line signals the beginning of a line-folded comment. Such a comment encompasses each subsequent line that starts with a comment, up to and including the first that does not have a backslash as the last non-blank character. The text – ignoring the initial hash characters – of a series of comments of this form is unfolded analogously to text field contents, with the unfolded result taken as the text of a single comment.

For example, applying comment unfolding to this CIF fragment:

```
# Non-folded comment
#\
# Folded comment 1 start... \
#folded comment 1 end
#\
# Folded comment 2 start...\
# folded comment 2 end\
_item value
# Comment 3
```

produces this:

```
# Non-folded comment
# Folded comment 1 start... folded comment 1 end
# Folded comment 2 start... folded comment 2 end
_item value
# Comment 3
```

Note that despite the trailing backslash in its folded form, the second folded comment ends with the line before the data item. It neither unfolds the item into the comment body nor continues into the comment following the item.

CIF processors are left at their discretion as to whether to apply unfolding to comments, but it should be noted that in both versions of the CIF syntax documented herein, a leading comment serves as a CIF-version comment only if it has the correct form without unfolding.

2.2.4.2.2.3 Line termination

CIF 2.0 syntax attributes identical significance to each of its three accepted line-termination sequences (§2.2.3.2.1), including when they appear inside data values, as if each line termination sequence other than a newline were converted to a newline prior to parsing. Therefore, the consumer of a multiline data value parsed from a CIF 2.0 file cannot expect to determine the form or forms of line terminators that appeared in the value's lexical representation. Where a processor presents such values to applications as character sequences with embedded line terminators, it should consistently use the same terminator. Absent a compelling reason to use something else, that terminator should be a single newline. It furthermore follows that CIF 2.0 producers cannot rely on conveying information by varying line terminators.

In contrast, CIF 1.1 relies on local text conventions for line termination semantics. Those can be understood as the convention or conventions used by default for handling text in the data and operating environment in which the CIF data are accessed. However, inasmuch as files may be shared among diverse systems and stored on network services accessible simultaneously to heterogeneous systems, and inasmuch as the standard text-handling conventions of some popular programming languages perform line-terminator translation, a processor that applied CIF 2.0 line-termination handling to input files could be regarded as interpreting those files according to local text conventions, as CIF 1.1 requires. Nevertheless, CIF 1.1 processors are not obligated to take this approach. The original view was that processors should be able simply to handle CIFs via whatever tools, facilities, and functions they ordinarily use for handling text files. Following that valid approach implies considering CIF 1.1 documents (more) environment-sensitive, and more likely to require conversion when moved from system to system or otherwise processed on a system different from the one on which they were produced. In effect, it narrows the scope of documents that can be considered valid CIFs in any given environment.

Additionally, in view of the record-oriented text file formats that are conventional in some

environments, and because of some programming languages' record-oriented text input / output interfaces and fixed-length character data types, some CIF 1.1 processors may be unable easily to recognize significant in-line whitespace at the ends of lines of a text field, and some CIF 1.1 writers may pad lines, including lines of text fields, with insignificant whitespace characters. Therefore, trailing whitespace in CIF 1.1 text field lines is considered insignificant. It should be elided where feasible, and if not elided, it should be ignored. Significant trailing whitespace can be marked and protected by use of the line-folding protocol. On the other hand, because CIF 2.0 is a binary byte stream format with explicit line termination sequences, no such considerations apply to multiline data values expressed in that syntax, neither text fields nor triple-quoted strings. Trailing whitespace on a line of a CIF 2.0 data value is significant, so removing it alters the meaning of the value. In view of CIF 1.1 behaviour, however, it is recommended that significant trailing whitespace be avoided in CIF 2.0 text fields. As with CIF 1.1, this can be achieved by applying the line-folding protocol if necessary.

2.2.4.2.2.4 Character and markup codes

The limited character set of CIF 1.1 poses a problem for conveying many kinds of data items relevant to CIF's intended area of use. Chemical names, personal and institutional names, chemical and mathematical formulae, non-English text, and even standard symbols for some relevant units of measure all may rely on characters outside CIF 1.1's set. So as to avoid defining mechanisms on a per-data-name basis for representing such characters, a set of conventional codes is defined for representing the additional characters most commonly needed in chemical and crystallographic texts written in Latin-based scripts. These codes are interpreted within those items whose definitions call for it, whether via a general rule applying to whole dictionaries or via explicit reference. Moreover, although CIF 2.0 syntax provides superior alternatives to almost all of the character codes described in this section, the syntax with which a CIF document is written does not replace or override the significance, if any, ascribed to these codes by data definitions. On the other hand, although these codes are recognized in many items' values, they should not be taken as universal. They are not interpreted in the values of items whose definitions do not call for it.

The first group of character codes provide for representing Greek letters. These have the form of a backslash character immediately followed by the Latin letter corresponding to the intended Greek one. These codes are case-sensitive: the Greek letter represented by one of these codes matches the case of the Latin letter in the code. The complete set is:

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

The second group of codes assist in conveying Latin letters bearing diacritical marks. They function similarly to Unicode combining characters in that they modify the meaning of an adjacent character, but unlike combining characters, these codes are presented immediately preceding the characters they modify, instead of immediately following. Each has the form of a backslash followed by a punctuation mark drawn from CIF 1.1's character set.

\'	acute (é)	\"	umlaut (ü)
\=	overbar / macron (ā)	\`	grave (à)
\~	tilde (ñ)	\.	overdot (ż)
\^	circumflex (ê)	\;	ogonek (ų)
\<	hacek / caron (ě)	\,	cedilla (ç)
\>	chuvash / double acute (ő)	\(	breve (ŏ)

For example, the character sequence \'e represents a lowercase 'e' with acute accent, 'é'. These codes are interpreted as such only when immediately followed by a Latin letter. They cannot be chained together.

There are several codes for specific letter forms not served by the combining codes:

\%a	å	\%A	Å	A with ring	\/l	ł	\/L	Ł	barred L
\/d	đ	\/D	Đ	barred D	\/o	ø	\/O	Ø	O with slash
\?i	ı	\?I	İ	dotting-variant I	\&s	ß	\&S	ẞ	sharp S

There are also codes for several non-letter symbols:

\%	degree[3] (°)	\\sim	~
+-	±	\\simeq	≃
-+	∓	\\neq	≠
--	dash	\\times	×
---	single bond[4]	\\square	superscript 2 (²)[5]
\\db	double bond[4]	\\langle	⟨
\\tb	triple bond[4]	\\rangle	⟩
\\ddb	delocalized double bond[4]	\\lightarrow	←
\\infty	∞	\\reftarrow	→

For the most part, these non-letter codes are sensitive to their trailing context. Specifically,

- those introduced by a doubled backslash are terminated by and include a following whitespace character;
- the character sequence \% is interpreted as part of the longer code given previously when the next character is an upper- or lowercase Latin A;
- the character sequence "--" represents a dash only when not part of a longer "---" sequence.

In addition to character codes, these conventions define typographic style codes for expressing superscript, subscript, italic, and boldface text. Each of these codes comprises both a starting and an ending delimiter for the text to which it applies, as shown below.

^super^	superscript	x^{super}
~sub~	subscript	y_{sub}
bold	bold typeface	**bold**
italic	italic typeface	*italic*

These specifications do not define the meaning of text in which these delimiters appear unpaired, or in which the regions marked by these codes overlap without one completely containing the other. The coincidence of the codes for italic and boldface type with HTML markup should not be taken as an indication that these conventions support any HTML features beyond those expressly documented here.

Overall, these codes and conventions are limited in their expressive ability. If it is necessary to convey more complex styling or typographic information then the entire text should be expressed as the value of an item having a content type supporting such information. Alternatively, as a replacement for most of the character codes, simply expressing the data set in CIF 2.0 syntax provides direct access to substantially all Unicode characters.

2.2.4.2.3 Numeric types

The conventional CIF-syntax representation of numeric types takes the form of an optionally-signed, floating-point, decimal number with optional power-of-ten scale and optional parenthesized uncertainty

3 Except when the next character is an upper- or lowercase 'a'.
4 Unicode defines no characters having significance for a chemical bond of any order, so this code does not have a corresponding Unicode character.
5 It is preferable to use style codes for this purpose: ^2^.

in the last digits, presented whitespace-delimited. When a number is expressed with both explicit scale and a standard uncertainty, the parenthesized uncertainty comes after the scale. For example:

CIF value	number	uncertainty
1085.3(3)	1085.3	0.3
10853e-01(3)	1085.3	0.3
+1.0853e3(30)	1085.3	3.0
-3e4(2)	-30000	20000
42	42	unspecified
3.14	3.14	unspecified

As the table indicates, omission of the standard uncertainty component does not necessarily imply that the value represents an exact number. That depends on the item, and as an external semantic consideration, such an omission may be acceptable, required, or forbidden for the values of specific data items. DDL2 dictionaries (ch x.x.x), including mmCIF (ch y.y.y), define separate items for measurands and their standard uncertainties. DDLm dictionaries implicitly define separate su items (see 2.x.x.x).

Formally, the complete syntax values are described by these EBNF productions; the *cif-number* symbol matches all value representations consistent with this type:

```
cif-number = cif-float, [ '(', cif-digits, ')' ]

cif-float = [ cif-sign ], ( ( cif-digits, [ '.', [ cif-digits ] ] ) | ( '.',
   cif-digits ) ), [ cif-exponent ] ;

cif-exponent = ( 'e' | 'E' ), [ cif-sign ], cif-digits ;

cif-sign = '+' | '-' ;

cif-digits = cif-digit, { cif-digits } ;

cif-digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
```

2.2.4.2.4 Implicit data typing

The original CIF 1.0 specifications describe an implicit data typing system for assigning either character, numeric, or null type to values based on the syntax with which they were expressed, similar to the syntactic typing described earlier in this section. Yet data values cannot be used for any practical purpose without knowing their definitions, and an item's definition specifies its (semantic) data type. Furthermore, CIF authors have not historically taken sufficient care to distinguish text values that take the form of numbers, such as the version code "2.0", from values that are truly numeric. Overall, it follows and is well established that semantic significance can be ascribed to whether a simple data value is presented whitespace-delimited *versus* in one of the several quoted forms. These specifications affirm that, and support it through the distinction between quoted and unquoted syntactic types. On the other hand, except as described in the preceding sections, these specifications expressly decline to

define what that semantic significance may be in any given case. In particular, whitespace-delimited data values should not be assumed numeric merely on the basis that they take the form described in section 2.2.4.2.3.

2.2.4.3 Alternative content types

Although initially designed with a view toward transferring and storing primarily numeric and plain textual information, CIF syntax has come to be used to store and convey information of a wide variety of forms and types. The character and markup codes presented in section 2.2.4.2.2.4 characterize an enriched content type suitable for expressing some items, and the wide character repertoire of CIF 2.0 makes it better adapted for names and text expressed in a variety of human languages and scripts. There is a wide variety of rich content types in modern use, however. These can be embedded directly in CIF data items whose definitions provide for that and where these specifications afford it. No general-purpose mechanism is defined for embedding objects of other types in CIF, but some special cases of that problem have been addressed, for example in CBF (see chapter XXX).

2.2.5 Considerations for CIF exchange and archiving

Both CIF syntax versions described in this chapter are designed to be independent of operating system (OS) and programming language, but they take different approaches to achieving that objective. CIF 1.1 is more abstract, relying on the ubiquity of text and text handling, and on common characteristics that it depends upon text implementations to provide, without regard to the specific implementation of those features. In this way, it is first and foremost the specification itself that is OS-independent. CIF 1.1 documents are OS-independent in the sense that their contents can be represented as an equivalent CIF 1.1 document on any OS that provides a minimal set of text features, but there is no expectation that representations of the same CIF 1.1 document in dissimilar operating environments will be or can be byte-for-byte identical. On the other hand, CIF 2.0 is more concrete. It relies on international standards (Unicode) and its own specific definitions (*e.g.* for line terminators) to describe CIF 2.0 documents in terms of sequences of bytes with OS-independent significance. The bytes forming a CIF 2.0 document therefore have exactly the same collective meaning in every environment, but they do not conform to some systems' expectations for text. Both formats are suited to exchanging and archiving information, but some of the considerations involved in doing so differ between them.

2.2.5.1 CIF 1.1 text

CIF 1.1 relies on a set of characters that all can be represented by the US-ASCII character encoding, but it does not rely on that encoding itself. CIF 1.1 documents may be constructed, manipulated, stored, and exchanged using character encodings that are incongruent with ASCII, including alternative single-byte encodings such as EBCDIC and multi-byte encodings such as UTF-16, provided only that such documents can be construed as conforming to local text conventions. Similarly, CIF 1.1 expects documents to be divided into lines in the conventional manner, whatever that happens to be.

Because CIF 1.1 does not actually define what local text conventions are, it may be practical to construe them on a per-collection basis, as the conventions employed by the one or more CIFs in that collection. Such an interpretation would imply that the applicable set of text conventions is an essential item of metadata that must be maintained and conveyed with every collection of CIF 1.1 documents. It would also place the onus on CIF consumers to employ the correct conventions for specific CIFs, either by using software and tools that support those conventions or by transforming CIFs to a supported set of conventions at need. This is effectively the current practice, though it is rarely formalized.

It should be emphasized that the various text conventions that can be employed do not confer any additional features on CIF. In particular, characters outside the CIF 1.1 character set must not appear literally in CIF 1.1 documents, regardless of any ability of local text conventions to express such characters.

2.2.5.2 CIF 2.0 documents

CIF 2.0 relies specifically on the UTF-8 character encoding, regardless of local conventions for text files, therefore CIF 2.0 documents are not appropriate inputs to some text-manipulation utilities in some environments. Furthermore, they should not, in general, be transcoded to a different character encoding, for if the result of such a transcoding differs from the original file then it is not a valid CIF 2.0 document. Moreover, depending on the CIF and the target encoding, transcoding may also corrupt the file's content. It is permissible, however, to convert among the line-termination sequences recognized by CIF 2.0 (see section 2.2.3.2.1). Doing so does not change the meaning of the file, and it may make CIF 2.0 documents more amenable for use with some of the utilities of the operating environment.

It is also permissible to prepend a single ***U+FEFF*** character to a CIF 2.0 file that does not already have one, as some Unicode-manipulation utilities are known to do, or to remove such a character. Those actions do not change the meaning of any file with respect to these specifications. However, because CIF 2.0 documents are permitted contain ***U+FEFF*** as their first character but not anywhere else, care must be exercised if multiple CIF 2.0 documents are concatenated. In that case, any leading ***U+FEFF*** character in the second and subsequent documents must be stripped or converted to whitespace lest the concatenated result fail to be a valid CIF 2.0 document.

References

Bernstein, H. J., Bollinger, J. C., Brown, I. D., Gražulis, S., Hester, J. R., McMahon, B., Spadaccini, N., Westbrook, J. D. & Westrip, S. P. (2016). *Specification of the Crystallographic Information File format, version 2.0*. J. Appl. Cryst. **49**, 277-284.

COMCIFS (2003). *CIF File Syntax Version 1.1 Working specification*. https://www.iucr.org/resources/cif/spec/version1.1/cifsyntax.

Hall, S. R. (1991). *The STAR file: a new format for electronic data transfer and archiving*. J. Chem. Inf.

Model. **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. & Spadaccini, N. (1994). *The STAR File: Detailed Specifications*. J. Chem. Inf. Model. **34**, 505–508.

International Standards Organization (1991). *ISO/IEC 646:1991 – Information Technology – ISO 7-bit coded character set for information interchange*. International Standards Organization, Geneva, Switzerland.

International Standards Organization (1996). *ISO/IEC 14977:1996 – Information Technology – Syntactic Metalanguage – Extended BNF*. International Standards Organization, Geneva, Switzerland.

Spadaccini, N. & Hall, S. R. (2012). *Extensions to the STAR File Syntax*. J. Chem. Inf. Model. **52**, 1901–1906.

The Unicode Consortium (2014*a*). *The Unicode Standard, Version 7.0.0*, ch. 3, §3.9. Mountain View: The Unicode Consortium. http://www.unicode.org/versions/Unicode7.0.0/.

The Unicode Consortium (2014*b*). *The Unicode Standard, Version 7.0.0*, ch. 3, §3.13. Mountain View: The Unicode Consortium. http://www.unicode.org/versions/Unicode7.0.0/.

Figure 1. A schematic view of the CIF data model. A CIF is made up of zero or more data blocks. Each data block contains zero or more loops and zero or more save frames, and each save frame contains zero or more loops. Each loop comprises one or more packets of one or more data items each. Each data item associates one data name with one data value. (Both CIF syntaxes described in this chapter permit single-packet loops to be expressed as series of separate data name / data value pairs.)

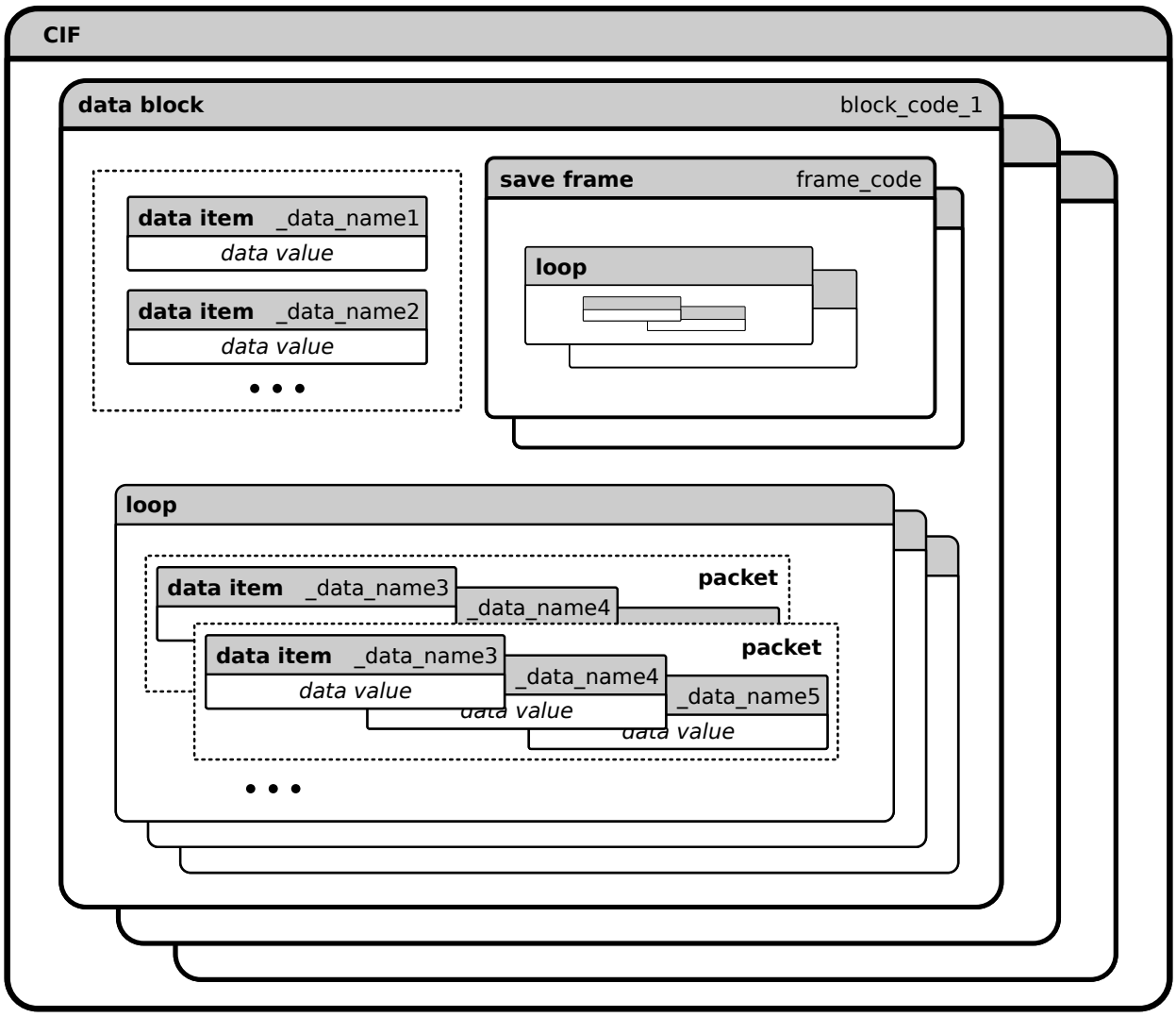

Figure 2. A simple CIF describing a small-molecule structure using syntax version 1.1 and data names from the DDLm version of the Core dictionary

```
data_sj13_025

_chemical.name_systematic          '1,2-napthoquinone'
_chemical.formula_sum              'C10 H6 O2'
_chemical.formula_weight           158.15

_symmetry.cell_setting             'monoclinic'
_symmetry.space_group_name_H-M     'P c'
_symmetry.space_group_name_Hall    'P -2yc'

loop_
 _symmetry_equiv.pos_as_xyz
 'x, y, z'
 'x, -y, z+1/2'

_cell.length_a                     3.7505(4)
_cell.length_b                     8.2550(8)
_cell.length_c                     11.7836(12)
_cell.angle_alpha                  90.00
_cell.angle_beta                   95.920(6)
_cell.angle_gamma                  90.00
_cell.volume                       362.88(6)
_cell.formula_units_Z              2

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
C1  C  0.0251(4)    0.7811(2)   -0.00226(16)  0.0147(4)
C2  C -0.1785(4)    0.6637(2)   -0.07803(15)  0.0135(4)
O3  O -0.2912(3)    0.69231(14) -0.17651(13)  0.0213(4)
C4  C -0.2603(5)    0.49549(19) -0.02786(16)  0.0166(4)
O5  O -0.4546(4)    0.40323(16) -0.08643(14)  0.0266(4)
C6  C -0.1043(4)    0.4589(2)    0.08722(16)  0.0163(4)
C7  C  0.0860(4)    0.5707(2)    0.15085(16)  0.0157(4)
C8  C  0.1489(4)    0.7344(2)    0.10965(14)  0.0139(4)
C9  C  0.3289(4)    0.8484(2)    0.18201(17)  0.0173(4)
C10 C  0.3831(5)    1.0053(2)    0.14351(17)  0.0194(4)
C11 C  0.2639(4)    1.0493(2)    0.03225(19)  0.0202(4)
C12 C  0.0858(5)    0.9371(2)   -0.04034(18)  0.0172(4)
```

Figure 3. A syntactically complete example CIF comprising excerpts of the DDLm version of the CIF Core dictionary, expressed in CIF 2.0 syntax

```
#\#CIF_2.0

data_CIF_CORE

_dictionary.title                       CIF_CORE
_dictionary.class                       Instance
_dictionary.version                     3.0.04
_dictionary.date                        2015-02-18
_dictionary.uri                         www.iucr.org/cif/dic/cif_core.dic
_dictionary.ddl_conformance             3.11.04
_dictionary.namespace                   CifCore
_description.text
;
    The CIF_CORE dictionary records all the CORE data items defined
    and used with in the Crystallographic Information Framework (CIF).
;

save_CIF_CORE

_definition.id                          CIF_CORE
_definition.scope                       Category
_definition.class                       Head
_definition.update                      2014-06-18
_description.text
;
     The CIF_CORE group contains the definitions of data items that
     are common to all domains of crystallographic studies.
;
_name.category_id                       CIF_DIC
_name.object_id                         CIF_CORE

save_

save__refln.hkl

_definition.id                          '_refln.hkl'
loop_
  _alias.definition_id
         '_refln.hkl'
_definition.update                      2012-11-26
_description.text
;
     The Miller indices as a reciprocal space vector.
;
_name.category_id                       refln
_name.object_id                         hkl
_type.purpose                           Number
_type.source                            Derived
_type.container                         Matrix
_type.contents                          Integer
_type.dimension [3]
loop_
  _method.purpose
```

```
  _method.expression
         Evaluation
;
     With r  as  refln

             _refln.hkl =  [r.index_h, r.index_k, r.index_l]
;

save_
```