

5.1. General considerations in programming CIF applications

BY H. J. BERNSTEIN

5.1.1. Introduction

There are many ways to create new ‘CIF-aware’ applications and to adapt existing applications to make them CIF-aware. This chapter reviews general considerations in programming CIF-aware applications, ranging from leaving an application CIF-unaware and relying on external filter utilities to do the job, through engineering an existing application to directly read and write CIFs, to writing a new CIF-aware application from scratch. The adaptation of applications to CIF does not happen in isolation. There are many other data representations and metadata frameworks relevant to crystallography. In Chapter 1.1, the CIF format was placed in the historical context of the development of data representation and metadata languages. In this chapter, we deal with that context from the perspective of software design.

The major issues in making an application CIF-aware are:

- (1) Are CIFs to be read?
 - (i) Are these CIFs produced externally?
 - (ii) Does the organization of information required by the application conform to the organization of information specified in the relevant dictionaries?
 - (iii) Is maximal performance in reading important?
- (2) Are CIFs to be written?
 - (i) Are these CIFs to be used externally?
 - (ii) Does the organization of information used by the application internally conform to the organization of information specified by the relevant dictionaries?
 - (iii) Is maximal performance in writing important?

Reading a CIF is a much more complex task than writing a CIF. Two equally valid CIF presentations of exactly the same information may be ordered differently. An application that reads a CIF must be prepared to accept tags and columns of tables in any order. The same order independence means that an application may write out tags and columns of tables in any convenient order, simplifying the design of CIF write logic.

When CIFs are only to be used internally, it is tempting to adjust the format to fit the application, *e.g.* by imposing order dependence. However, caution is advised if there is any possibility that such an application might eventually have to deal with CIFs coming from or going to other groups. In designing the CIF interface for an application, it is prudent to assume that the read logic will eventually have to deal with externally produced CIFs and that CIFs produced by the write logic will eventually be processed by software at other sites.

If performance is not a major issue, then it can be easy to make an application CIF-aware simply by the use of external filter programs. However, when performance *is* an issue, it becomes necessary to integrate the CIF reading and writing logic with the application. This can be done with reasonable efficiency by the use of existing CIF-aware libraries, but such libraries can impose a cost by their use of private internal data structures to hold the

information from a CIF. Integrated design from scratch may be needed for maximal performance.

Creating or adapting software for CIF is an example of creating or adapting software for an agreed format, a format to be adhered to in the creation of multiple applications. There are different levels of ‘agreement’. The agreement may apply to the representation of data, the representation of information about data (‘meta-data’) or both (making a ‘data framework’). The agreement may loosely specify the style of presentation of information or may specify details of presentation with great precision, or anything in between. The effort one needs to make in adapting an application to an agreed format depends to a large part on the level of agreement. If the agreed format is sufficiently detailed, one can comply strictly with the agreed format as a standard.

If one’s goals are the widest possible interchange of data and the longest possible survival time of data in archives, it is important to achieve strict adherence to the agreed format as a standard. If one’s goals are shorter-term and do not raise issues of interchange with independent groups, use of an agreed format as a checklist or style guide may help to avoid redundant effort. Even when one has longer-term goals, system requirements may not mesh with the agreed format and the development of new formats may be needed. CIF, as an agreed format, involves specification of metadata for a specific data format and of ontologies of domain-specific definitions that could be used not only in CIF but also in other formats. The use of an agreed format as a base can help to avoid redundant effort in this case as well. Within the domain of small-molecule crystallography, CIF has achieved its powerful impact on crystallographic publication and archiving by being used as a strict standard both for the data format and for definitions of terms. Within other domains or for certain applications other approaches may be appropriate. For example, it has proven productive to make use of CIF data names within XML (Bray *et al.*, 1998) formatted documents (Bernstein & Bernstein, 2002).

5.1.2. Background

There have been many efforts at creating agreed formats for data to be used in crystallography (see Chapter 1.1). We need to consider how software has been created to make use of such formats, especially software to make use of CIF.

Agreement on formats evolved from the earliest efforts at collaboration among research groups. Within crystallography, recognition of the need to use data formats as standards and to adapt applications to agreed formats, rather than to adapt formats to the caprices of particular applications or diffractometers or graphics engines, began in the late 1960s and early 1970s with the establishment of computerized data resources for the chemical and crystallographic community and the increasing availability of computer networks (Lykos, 1975). We will discuss three early data-resource efforts: the Cambridge Crystallographic Data Centre Structural Database File (CSD) (Allen *et al.*, 1973), the Brookhaven National Laboratory Protein Data Bank (PDB) (Bernstein *et al.*, 1977) and the NIH/EPA Chemical Information System (CIS) (Heller *et al.*, 1977). The differences and similarities among application development efforts related to these resources illustrate some of the issues

Affiliation: HERBERT J. BERNSTEIN, Department of Mathematics and Computer Science, Kramer Science Center, Dowling College, Idle Hour Blvd, Oakdale, NY 11769, USA.

5. APPLICATIONS

that now face software developers working with CIF: conformance to agreed formats *versus* deviations from standards to improve performance, as well as cross-platform portability.

The Cambridge Crystallographic Data Centre was established in 1965 'to compile a database containing comprehensive information on small-molecule crystal structures, *i.e.* organics and metallo-organic compounds containing up to 500 non-H atoms, the structures of which had been determined by X-ray or neutron diffraction' (Allen, 2002). The Protein Data Bank was established at Brookhaven National Laboratory in 1971 as an archive of macromolecular structural information. The NIH/EPA Chemical Information System was established in 1975 as a confederation of databases including mass spectroscopy, NMR and the data from the CSD. The three resources, CSD, PDB and CIS, took different approaches to applications development. The CSD was an integrated software system centred on a database. Both the software and the database were distributed on magnetic tape for users to use on their local computers. The developers of the software had to be concerned with portability of the software across the multiple computer systems used by crystallographers, but retained control of the design of the retrieval software and a core suite of applications. The PDB was an archive, rather than a database. Some software and the data were distributed on magnetic tape, but the application development model was what would now be called 'open', with users and software developers taking the data and the PDB format specification and creating software that would do useful things with PDB entries. The CIS was a remotely accessed confederation of databases on a central computer. The developers of software for the CIS did not have to be concerned with cross-platform portability, or with changes in syntax or semantics of data files impacting on external software developers. Developers of software for the CSD and the PDB had to be concerned with strict compliance with the rules for the respective data formats, albeit on somewhat different timescales. Developers of software for the centralized CIS database could negotiate for immediate changes in the data format to improve performance of the relevant application.

The CSD had agreed internal formats (Cambridge Structural Database, 1978). However, as noted in Chapter 1.1, there were many different formats in use for small-molecule crystallography and related fields. One may conjecture that one of many causes for such divergence was the CCDC practice of acquiring much of its data from journals, after differences among data formats had been masked by the publication process. The transition from this Tower of Babel to CIF is described in Chapter 1.1, and that history will not be repeated here, but it is important to note that an application writer working in the domain of small-molecule crystallography still has to be aware of a wide variety of formats in addition to CIF.

In the beginning, the PDB went through a relatively rapid format change and then achieved a stable format for more than two decades. The PDB differed from the CSD in depending on user deposition of data prior to publication. The better a user conformed to PDB data-format conventions, the more efficiently could the data move from deposition to release. The initial standard PDB format (PDB, 1974) was derived from the format used in a popular refinement program of the day (Diamond, 1971) and used 132-character records identified by the character strings in the first six columns. Starting in 1976, the PDB spent more than a year (PDB, 1976*a,b*, 1977) converting to an 80-column format, extensions of which are still in use to this day. Many external programs were developed using this 80-column format and it has become a major *de facto* standard for macromolecular software applications. Most application packages producing crystallographic macromolecular

structures made a gradual transition from having output options for producing 'Diamond format' to having output options for producing PDB format. Macromolecular applications working with other disciplines shared the small-molecule applications penchant for multiple formats.

The CIS, working in a completely closed, central service environment, had little direct impact on the formats to be used for applications. The CIS would acquire data from existing archives and databases and meld them into its master database. It would deliver its data as text on a CRT. Much of the impact of CIS data formats was to be restricted to its own internal application development.

Most of the formats resulting from these early efforts were fixed-field, fixed-order formats. The result was that adapting an application to a data format was simple if the processing flow of the application conformed to the fixed order of the data format. Frequently, the data flow did conform. When the processing flow did not conform, it was necessary to create internal data structures or temporary files to allow the unfortunately timed arrival of data to be time-shifted until it was needed. In general, the heaviest burden was imposed on applications that needed to write data conforming to one of the agreed formats. As the complexity of such time-shifting processes increased, it became clear that the cleanest solution was to base an application on an internal database and to populate the database as the data were processed. When data were to be written by an application, the data could be extracted from the database in whatever order was required.

In the 1970s and early 1980s, such a procedure was a serious burden to place on an application. With limited memory and processor speeds, there was a strong argument for adapting agreed formats to the 'natural' processing flow, reducing or avoiding the need for an internal database. As the speed and size of computers have changed and as programming language and operating-system support for dynamic allocation of resources has improved, the need to have agreed formats driven by applications has become less pressing.

We need to understand three major thrusts in data representation: the development of markup languages, of data-representation frameworks and of database application support. Modern applications can benefit from all three.

5.1.2.1. Markup languages

A markup language allows the raw text of a document to be annotated with interleaved 'markup' specifying layout information for the bracketed text. For document processing, the implicit assumption of the use of an internal database became formalized with the gradual adoption of agreed markup languages in the late 1980s and early 1990s [*e.g.* \TeX (Knuth, 1986), SGML (ISO, 1986), RTF (Andrews, 1987), HTML (Berners-Lee, 1989)]. When used in this manner, such a language has the implicit ordering assumption of reading forward in the document. However, with modern demands for multidimensional layout and document reflow, applications managing such documents achieve the best performance and flexibility when they store the entire marked-up document in an internal data structure that allows random access to all the information.

5.1.2.2. Data-representation frameworks

A data-representation framework provides the concepts for managing data and data about the management of data ('meta-data'). Such frameworks may be based on programming languages or markup languages or built from scratch. They provide a mechanism for representing data (*e.g.* as data sets, graphs or trees)

5.1. GENERAL CONSIDERATIONS IN PROGRAMMING CIF APPLICATIONS

and a mechanism for representing metadata (*e.g.* as dictionaries or schemas). Four are of particular importance in crystallography: CIF, ASN.1, HDF and XML.

As noted in Chapter 1.1, CIF was created to rationalize the publication process for small molecules. It combines a very simple tag–value data representation with a dictionary definition language (DDL) and well populated dictionaries. CIF is table-oriented, naturally row-based, has case-insensitive tags and allows two levels of nesting. CIF is order-independent and uses its dictionaries both to define the meanings of its tags and to parameterize its tags. It is interesting to note that, even though CIF is defined as order-independent, it effectively fills the role of an order-dependent markup language in the publication process. We will discuss this issue later in this chapter.

Abstract Syntax Notation One (ASN.1) (Dubuisson, 2000; ISO, 2002) was developed to provide a data framework for data communications, where great precision in the bit-by-bit layout of data to be seen by very different systems is needed. Although targeted for communications software, ASN.1 is suitable for any application requiring precise control of data structures and, as such, primarily supports the metadata of an application, rather than the data. ASN.1 can be compiled directly to C code. The resulting C code then supports the data of the application. ASN.1 notation found application in NCBI’s macromolecular modelling database (Ohkawa *et al.*, 1995). ASN.1 has case-sensitive tags and allows case-insensitive variants. It manages order-dependent data structures in a mixed order-dependent/order-independent environment.

HDF (NCSA, 1993) is ‘a machine-independent, self-describing, extendible file format for sharing scientific data in a heterogeneous computing environment, accompanied by a convenient, standardized, public domain I/O library and a comprehensive collection of high quality data manipulation and analysis interfaces and tools’ (<http://ssdoo.gsfc.nasa.gov/nost/formats/hdf.html>). HDF was adopted by the Neutron and X-ray Data Format (NeXus) effort (Klosowski *et al.*, 1997). HDF allows the building of a complete data framework, representing both data and metadata. Two parallel threads of software development, focused on the management and exchange of raw data from area detectors, began in the mid-1990s: the Crystallographic Binary File (CBF) (Hammersley, 1997) and NeXus. The volumes of data involved were daunting and efficiency of storage was important. Therefore both proposed formats assumed a binary format. CBF was based on a combination of CIF-like ASCII headers with compressed binary images. NeXus was based on HDF. The first API for CBF was produced by Paul Ellis in 1998. CBF rapidly evolved into CBF/imgCIF with a complete DDL2 dictionary and a fully CIF-compliant API (Chapter 5.6). As of mid-2010, NeXus was still evolving (see <http://www.nexusformat.org/>).

XML is a simplified form of SGML, drawing on years of development of tools for SGML and HTML. XML is tree-oriented with case-sensitive entity names. It allows unlimited nesting and is order-dependent. Metadata are managed as a ‘document type definition’ (DTD), which provides minimal syntactic information, or as schemas, which allow for more detail and are more consistent with database conventions. In fields close to crystallography, the first effort at adopting XML was the chemical markup language (CML) (Murray-Rust & Rzepa, 1999). CML is intentionally imprecise in its ontology to allow for flexibility in development. The CSD and PDB have released their own XML representations (http://www.ccdc.cam.ac.uk/support/documentation/relibase/3_0/relibase_DPG/toc.html; <http://pdbml.rcsb.org>).

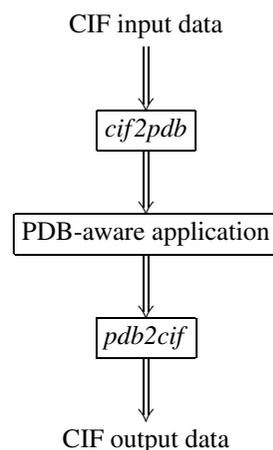


Fig. 5.1.3.1. Example of using filters to make a PDB-aware application CIF-aware.

It may seem from this discussion that the application designer faces an unmanageable variety of data frameworks in an unstable, evolving environment. To some extent this is true. Fortunately, however, there are signs of convergence on CIF dictionary-based ontologies and the use of transliterated CIFs. This means that an application adapted to CIF should be relatively easy to adapt to other data frameworks.

5.1.3. Strategies in designing a CIF-aware application

There are multiple strategies to consider when designing a CIF-aware application. One can use external filters. One can use existing CIF-aware libraries. One can write CIF-aware code from scratch.

5.1.3.1. Working with filter utilities

One solution to making an existing application aware of a new data format is to leave the application unchanged and change the data instead. For almost all crystallographic formats other than CIF, the Swiss-army knife of conversion utilities is *Babel* (Walters & Stahl, 1994). *Babel* includes conversions to and from PDB format. Therefore, by the use of *cif2pdb* (Bernstein & Bernstein, 1996) and *pdb2cif* (Bernstein *et al.*, 1998) combined with *Babel*, many macromolecular applications can be made CIF-aware without changing their code (see Figs. 5.1.3.1 and 5.1.3.2). If the need is to extract mmCIF data from the output of a major application, the PDB provides *PDB_EXTRACT* (http://sw-tools.pdb.org/apps/PDB_EXTRACT/).

Creating a filter program to go from almost any small-molecule format to core CIF is easy. In many cases one need only insert the appropriate ‘`loop_`’ headers. Creating a filter to go from CIF to a particular small-molecule format can be more challenging, because a CIF may have its data in any order. This can be resolved by use of *QUASAR* (Hall & Sievers, 1993) or *cif2cif* (Bernstein, 1997), which accept request lists specifying the order in which data are to be presented (see Fig. 5.1.3.3).

There are a significant and growing number of filter programs available. Several of them [*QUASAR*, *cif2cif*, *ciftex* (<ftp://ftp.iucr.org/pub/ciftex.tar.Z>) (to convert from CIF to \TeX) and *ZINC* (Stampf, 1994) (to unroll CIFs for use by Unix utilities)] are discussed in Chapter 5.3. In addition there are *CIF2SX* by Louis J. Farrugia (<http://www.chem.gla.ac.uk/~louis/software/utis/>), to convert from CIF to *SHELXL* format, and *DIFRAC* (Flack *et al.*, 1992) to translate many diffractometer output formats to CIF. The program *cif2xml* (Bernstein & Bernstein, 2002) translates from CIF to XML and CML. The PDB

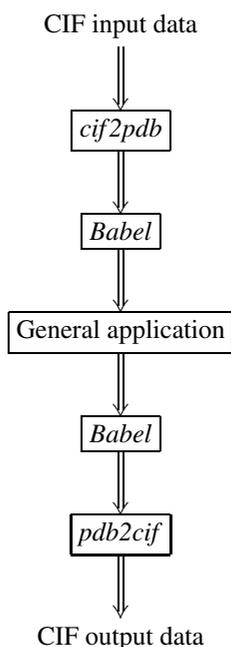


Fig. 5.1.3.2. Example of using filters to make a general application CIF-aware.

provides *CIFTr* by Zukang Feng and John Westbrook (<http://sw-tools.pdb.org/apps/CIFTr/>) to translate from the extended mmCIF format described in Appendix 3.6.2 to PDB format and *MAXIT* (<http://sw-tools.pdb.org/apps/MAXIT/>), a more general package that includes conversion capabilities. See also Chapter 5.5 for an extended discussion of the handling of mmCIF in the PDB software environment.

5.1.3.2. Using existing CIF libraries and APIs

Another approach to making an existing application CIF-aware or to design a new CIF-aware application is to make use of one (or more) of the existing CIF libraries and application programming interfaces (APIs). Because the data involved need not be reprocessed, code that uses a library directly is often faster than equivalent code working with filter programs. The code within an application can be tuned to the internal data structures and coding conventions of the application.

The approach to internal design depends on the language, data structures and operating environment of the application. A few years ago, the precise details of language version and operating system would have been major stumbling blocks to conversion. Today, however, almost every platform supports a variation

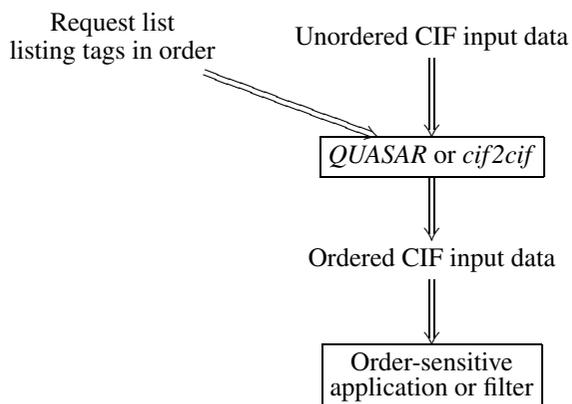


Fig. 5.1.3.3. Using *QUASAR* or *cif2cif* to reorder CIF data for an order-dependent application or filter.

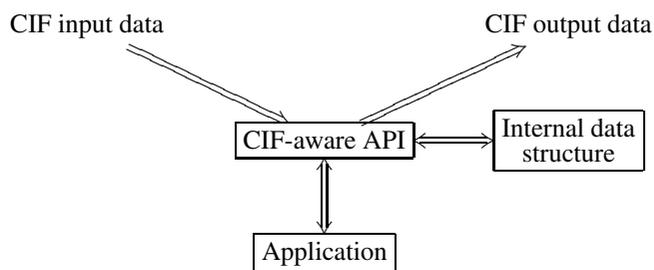


Fig. 5.1.3.4. Typical dataflow of a C-based CIF API.

of the Unix application programming interface and many languages have viable interfaces to C and/or C++. Therefore it is often feasible to consider use of C, C++ or Objective-C libraries, even for Fortran applications. *Star_Base* (Spadaccini & Hall, 1994; Chapter 5.2) is a program for extracting data from STAR Files. It is written in ANSI C and includes the code needed to parse a STAR File. *OOSTAR* (Chang & Bourne, 1998; Chapter 5.2) is an Objective-C package that includes another parser for STAR Files (<http://www.sdsc.edu/pb/cif/OOSTAR.html>). *CIFLIB* (Westbrook *et al.*, 1997) provides a CIF-specific API. *CIFPARSE* (Tosic & Westbrook, 1998) is another C-based library for CIF. *CBFlib* (Chapter 5.6) is an ANSI C API for both CIF and CBF/imgCIF files. The *CifSieve* package (Hester & Okamura, 1998) provides specialized code generation for retrieval of particular data items in either C or Fortran (see Chapter 5.3 for more details). The package *cciflib* (Keller, 1996) (<http://www.ccp4.ac.uk/dist/html/mmCIFformat.html>) is used by the *CCP4* program suite to support mmCIF in both C and Fortran applications. If an application in Fortran is to be converted with a purely Fortran-based library, the package *CIFtbx* (Hall, 1993; Hall & Bernstein, 1996) is a solution. See Chapter 5.4 for more details.

The common interface provided in C-based applications is for the library to buffer the entire CIF file into an internal data structure (usually a tree), essentially creating a memory-resident database (see Fig. 5.1.3.4). This preload greatly reduces any demands on the application to deal with the order-independence of CIF, at the expense of what can be a very high demand for memory. The problem of excessive memory demand is dealt with in *CBFlib* by keeping large text fields on disk, with only pointers to them in memory. In some libraries, validation of tags against dictionaries is handled by the API. In others it is the responsibility of the application programmer. While the former approach helps to catch errors early, the second, 'lightweight' approach is more popular when fast performance is required.

The most commonly used versions of Fortran do not include dynamic memory management. In order to preload an arbitrary CIF, one needs to use one of the C-based libraries. Alternatively, a pure Fortran application can transfer CIFs being read to a disk-based random access file. *CIFtbx* does this each time it opens a CIF. The user never works directly with the original CIF data set. This provides a clean and simple interface for reading, but slows all read access to CIFs. In Fortran, compromises are often necessary, with critical tables handled in memory rather than on disk, but this may force changes in dimensions and then recompilation when dictionaries or data sets become larger than anticipated.

5.1.3.3. Creating a CIF-aware application from scratch

The primary disadvantage of using an existing CIF library or API in building an application is that there can be a loss of performance or a demand for more resources than may be needed. The common practice followed by most libraries of building and

5.1. GENERAL CONSIDERATIONS IN PROGRAMMING CIF APPLICATIONS

```

cbf:      datablock      { cbf_failnez (cbf_find_parent (&($$), $1, CBF_ROOT)) }
;
cbfstart:
;
datablockstart:  cbfstart      { cbf_failnez (cbf_make_child (&($$), $1, CBF_DATABLOCK, NULL)) }
| cbf datablockname { cbf_failnez (cbf_make_child (&($$), $1, CBF_DATABLOCK, $2)) }
;
datablock:      datablockstart { $$ = $1; }
| assignment    { cbf_failnez (cbf_find_parent (&($$), $1, CBF_DATABLOCK)) }
| loopassignment { cbf_failnez (cbf_find_parent (&($$), $1, CBF_DATABLOCK)) }
;
category:      datablock categoryname { cbf_failnez (cbf_make_child (&($$), $1, CBF_CATEGORY, $2)) }
;
column:        category columnname { cbf_failnez (cbf_make_child (&($$), $1, CBF_COLUMN, $2)) }
| datablock itemname { cbf_failnez (cbf_make_new_child (&($$), $1, CBF_CATEGORY, NULL)) }
|                cbf_failnez (cbf_make_child (&($$), $$, CBF_COLUMN, $2)) }
;
assignment:    column value      { $$ = $1;
|                                cbf_failnez (cbf_set_columnrow ($$, 0, $2, 1)) }
;
loopstart:     datablock loop    { cbf_failnez (cbf_make_node (&($$), CBF_LINK, NULL, NULL))
|                                cbf_failnez (cbf_set_link ($$, $1)) }
;
loopcategory:  loopstart categoryname { cbf_failnez (cbf_make_child (&($$), $1, CBF_CATEGORY, $2))
|                                cbf_failnez (cbf_set_link ($1, $$))
|                                $$ = $1; }
| loopcolumn categoryname { cbf_failnez (cbf_find_parent (&($$), $1, CBF_DATABLOCK))
|                                cbf_failnez (cbf_make_child (&($$), $$, CBF_CATEGORY, $2))
|                                cbf_failnez (cbf_set_link ($1, $$))
|                                $$ = $1; }
;
loopcolumn:    loopstart itemname { cbf_failnez (cbf_make_new_child (&($$), $1, CBF_CATEGORY, NULL))
|                                cbf_failnez (cbf_make_child (&($$), $$, CBF_COLUMN, $2))
|                                cbf_failnez (cbf_set_link ($1, $$))
|                                cbf_failnez (cbf_add_link ($1, $$))
|                                $$ = $1; }
| loopcolumn itemname { cbf_failnez (cbf_find_parent (&($$), $1, CBF_DATABLOCK))
|                                cbf_failnez (cbf_make_child (&($$), $$, CBF_CATEGORY, NULL))
|                                cbf_failnez (cbf_make_child (&($$), $$, CBF_COLUMN, $2))
|                                cbf_failnez (cbf_set_link ($1, $$))
|                                cbf_failnez (cbf_add_link ($1, $$))
|                                $$ = $1; }
| loopcategory columnname { cbf_failnez (cbf_make_child (&($$), $1, CBF_COLUMN, $2))
|                                cbf_failnez (cbf_set_link ($1, $$))
|                                cbf_failnez (cbf_add_link ($1, $$))
|                                $$ = $1; }
;
loopassignment: loopcolumn value { $$ = $1;
|                                cbf_failnez (cbf_shift_link ($$))
|                                cbf_failnez (cbf_add_columnrow ($$, $2)) }
| loopassignment value { $$ = $1;
|                                cbf_failnez (cbf_shift_link ($$))
|                                cbf_failnez (cbf_add_columnrow ($$, $2)) }
;
loop:          LOOP
;
datablockname: DATA          { $$ = $1; }
;
categoryname:  CATEGORY       { $$ = $1; }
;
columnname:    COLUMN         { $$ = $1; }
;
itemname:      ITEM           { $$ = $1; }
;
value:         STRING         { $$ = $1; }
| WORD         { $$ = $1; }
| BINARY      { $$ = $1; }
;

```

Fig. 5.1.3.5. Example of *bison* data defining a CIF parser (taken from *CBFlib*).

preloading an internal data structure that holds the entire CIF may not be the optimal choice for a given application. When reading a CIF it is difficult to avoid the need for extra data structures to resolve the issue of CIF order independence. However, when writing data to a CIF, it may be sufficient simply to write the necessary

tags and values from the internal data structures of an application, rather than buffering them through a special CIF data structure.

It is tempting to apply the same reasoning to the reading of CIF and create a fixed ordering in which data are to be processed, so that no intermediate data structure will be needed to buffer a CIF.

5. APPLICATIONS

Unless the application designer can be certain that externally produced CIFs will never be presented to the application, or will be filtered through a reordering filter such as *QUASAR* or *cif2cif*, working with CIFs in an order-dependent mode is a mistake.

Because of the importance of being able to accept CIFs written by any other application, which may have written its data in a totally different order than is expected, it is a good idea to make use of one of the existing libraries or APIs if possible, unless there is some pressing need to do things differently.

If a fresh design is needed, *e.g.* to achieve maximal performance in a time-critical application, it will be necessary to create a CIF parser to translate CIF documents into information in the internal data structures of the application. In doing this, the syntax specification of the CIF language given in Chapter 2.2 should be adhered to precisely. This result is most easily achieved if the code that does the parsing is generated as automatically as possible from the grammar of the language. Current 'industrial' practice in creating parsers is based on use of commonly available tools for lexical scanning of tokens and parsing of grammars based on *lex* (Lesk & Schmidt, 1975) and *yacc* (Johnson, 1975). Two accessible descendants of these programs are *flex* (by V. Paxson *et al.*) and *bison* (by R. Corbett *et al.*). See Fig. 5.1.3.5 for an example of *bison* data in building a CIF parser. Both *flex* and *bison* are available from the GNU project at <http://www.gnu.org>.

Neither *flex* nor *bison* is used directly by the final application. Each may be used to create code that becomes part of the application. For example, both are used by *CifSieve* to generate the code it produces. There is an important division of labour between *flex* and *bison*; *flex* is used to produce a lexicographic scanner, *i.e.* code that converts a string of characters into a sequence of 'tokens'. In CIF, the important tokens are such things as tags and values and reserved words such as `loop_`. Once tokens have been identified, responsibility passes to the code generated by *bison* to interpret. In practice, because of the complexities of context-sensitive management of white space to separate tokens and the small number of distinct token types, *flex* is not always used to generate the lexicographic scanner for a CIF parser. Instead, a hand-coded lexer might be used.

The parser generated by *bison* uses a token-based grammar and actions to be performed as tokens are recognized. There are two major alternatives to consider in the design: event-driven interaction with the application or building of a complete data structure to hold a representation of the CIF before interaction with the application. The advantage of the event-driven approach is that a full extra data structure does not have to be populated in order to access a few data items. The advantage of building a complete representation of the CIF is that the application does not have to be prepared for tags to appear in an arbitrary order.

5.1.4. Conclusion

Making CIF-aware applications is a demanding, but manageable, task. A software developer has the choice of using external filters, using existing libraries and APIs, or of building CIF infrastructure from scratch. The last choice presents an opportunity to tune the handling of CIFs to the needs of the application, but also presents the risk of creating code that does not conform to CIF specifications. One can never know for certain how a new application may be used in the future. If there is any doubt that an application built from scratch will conform to CIF specifications, prudence dictates that one should use filter programs or well tested libraries and APIs in preference to cutting corners in building an application from scratch.

We are grateful to Frances C. Bernstein for her helpful comments and suggestions.

References

- Allen, F. H. (2002). *The Cambridge Structural Database: a quarter of a million crystal structures and rising*. *Acta Cryst.* **B58**, 380–388.
- Allen, F. H., Kennard, O., Motherwell, W. D. S., Town, W. G. & Watson, D. G. (1973). *Cambridge Crystallographic Data Centre. II. Structural Data File*. *J. Chem. Doc.* **13**, 119–123.
- Andrews, N. (1987). *Rich Text Format standard makes transferring text easier*. *Microsoft Syst. J.* **2**, 63–67.
- Berners-Lee, T. (1989). *Information management: a proposal*. Internal Report. Geneva: CERN. <http://www.w3.org/History/1989/proposal-msw.html>.
- Bernstein, F. C. & Bernstein, H. J. (1996). *Translating mmCIF data into PDB entries*. *Acta Cryst.* **A52** (Suppl.), C-576.
- Bernstein, F. C., Koetzle, T. F., Williams, G. J. B., Meyer, E. F. Jr, Brice, M. D., Rodgers, J. R., Kennard, O., Shimanouchi, T. & Tasumi, M. (1977). *The Protein Data Bank: a computer-based archival file for macromolecular structures*. *J. Mol. Biol.* **112**, 535–542.
- Bernstein, H. J. (1997). *cif2cif – CIF copy program*. Bernstein + Sons, Bellport, NY, USA. Included in <http://www.bernstein-plus-sons.com/software/cif2cif>.
- Bernstein, H. J. & Bernstein, F. C. (2002). *YAXDF and the interaction between CIF and XML*. *Acta Cryst.* **A58** (Suppl.), C257.
- Bernstein, H. J., Bernstein, F. C. & Bourne, P. E. (1998). *CIF applications. VIII. pdb2cif: translating PDB entries into mmCIF format*. *J. Appl. Cryst.* **31**, 282–295. Software available from <http://www.bernstein-plus-sons.com/software/pdb2cif>.
- Bray, T., Paoli, J. & Sperberg-McQueen, C. (1998). *Extensible Markup Language (XML)*. W3C recommendation 10-February-1998. <http://www.w3.org/TR/1998/REC-xml-19980210>.
- Cambridge Structural Database (1978). *Cambridge Crystallographic Database User Manual*. Cambridge Crystallographic Data Centre, 12 Union Road, Cambridge, England.
- Chang, W. & Bourne, P. E. (1998). *CIF applications. IX. A new approach for representing and manipulating STAR files*. *J. Appl. Cryst.* **31**, 505–509.
- Diamond, R. (1971). *A real-space refinement procedure for proteins*. *Acta Cryst.* **A27**, 436–452.
- Dubuisson, O. (2000). *ASN.1 – communication between heterogeneous systems*. San Francisco, CA: Morgan Kaufmann. (Translated from the French by P. Fouquart.)
- Flack, H. D., Blanc, E. & Schwarzenbach, D. (1992). *DIFRAC, single-crystal diffractometer output-conversion software*. *J. Appl. Cryst.* **25**, 455–459.
- Hall, S. R. (1993). *CIF applications. IV. CIFtbx: a tool box for manipulating CIFs*. *J. Appl. Cryst.* **26**, 482–494.
- Hall, S. R. & Bernstein, H. J. (1996). *CIF applications. V. CIFtbx2: extended tool box for manipulating CIFs*. *J. Appl. Cryst.* **29**, 598–603.
- Hall, S. R. & Sievers, R. (1993). *CIF applications. I. QUASAR: for extracting data from a CIF*. *J. Appl. Cryst.* **26**, 469–473.
- Hammersley, A. P. (1997). *FIT2D: an introduction and overview*. ESRF Internal Report ESRF97HA02T. Grenoble: ESRF.
- Heller, S. R., Milne, G. W. A. & Feldmann, R. J. (1977). *A computer-based chemical information system*. *Science*, **195**, 253–259.
- Hester, J. R. & Okamura, F. P. (1998). *CIF applications. X. Automatic construction of CIF input functions: CifSieve*. *J. Appl. Cryst.* **31**, 965–968.
- ISO (1986). ISO 8879. *Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. Geneva: International Organization for Standardization.
- ISO (2002). ISO/IEC 8824-1. *Abstract Syntax Notation One (ASN.1). Specification of basic notation*. Geneva: International Organization for Standardization.
- Johnson, S. C. (1975). *YACC: Yet Another Compiler-Compiler*. Bell Laboratories Computing Science Technical Report No. 32. Bell Laboratories, Murray Hill, New Jersey, USA. (Also in UNIX Programmer's Manual, Supplementary Documents, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.)
- Keller, P. A. (1996). *A mmCIF toolbox for CCP4 applications*. *Acta Cryst.* **A52** (Suppl.), C-576.

5.1. GENERAL CONSIDERATIONS IN PROGRAMMING CIF APPLICATIONS

- Klosowski, P., Koennecke, M., Tischler, J. Z. & Osborn, R. (1997). *NeXus: a common format for the exchange of neutron and synchrotron data*. *Physica B Condens. Matter*, **B241–243**, 151–153.
- Knuth, D. E. (1986). *The TeXbook. Computers and typesetting*, Vol. A. Reading, MA: Addison-Wesley.
- Lesk, M. E. & Schmidt, E. (1975). *Lex – a lexical analyzer generator*. Bell Laboratories Computing Science Technical Report No. 39. Bell Laboratories, Murray Hill, New Jersey, USA. (Also in UNIX Programmer's Manual, Supplementary Documents, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.)
- Lykos, P. (1975). Editor. *Computer networking and chemistry*. ACS Symposium Series, Vol. 19. Washington DC: American Chemical Society.
- Murray-Rust, P. & Rzepa, H. (1999). *Chemical markup, XML and the WWW, Part I: Basic principles*. *J. Chem. Inf. Comput. Sci.* **39**, 928–942.
- NCSA (1993). *NCSA HDF: specification and developer's guide*. Version 3.2. University of Illinois at Urbana-Champaign, USA.
- Ohkawa, H., Ostell, J. & Bryant, S. (1995). *MMDB: an ASN.1 specification for macromolecular structure*. In *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology, Cambridge, England, 16–19 July 1995*, pp. 259–267. Menlo Park, CA: American Association for Artificial Intelligence.
- PDB (1974). *PDB Newsletter 1*. Brookhaven National Laboratory, USA.
- PDB (1976a). *PDB Newsletter 2*. Brookhaven National Laboratory, USA.
- PDB (1976b). *PDB Newsletter 3*. Brookhaven National Laboratory, USA.
- PDB (1977). *PDB Newsletter 4*. Brookhaven National Laboratory, USA.
- Spadaccini, N. & Hall, S. R. (1994). *Star_Base: accessing STAR File data*. *J. Chem. Inf. Comput. Sci.* **34**, 509–516.
- Stampf, D. R. (1994). *ZINC – galvanizing CIF to work with UNIX*. Manual. Protein Data Bank, Brookhaven National Laboratory, USA.
- Tosic, O. & Westbrook, J. D. (1998). *CIFPARSE: A library of access tools for mmCIF. Reference guide*. Version 3.1. Nucleic Acid Database Project, Department of Chemistry and Chemical Biology, Rutgers, The State University of New Jersey, USA. <http://sw-tools.pdb.org/apps/CIFPARSE/cifparse/cifparse.html>.
- Walters, P. & Stahl, M. (1994). *BABEL reference manual*. Version 1.06. Dolata Research Group, Department of Chemistry, University of Arizona, USA.
- Westbrook, J. D., Hsieh, S.-H. & Fitzgerald, P. M. D. (1997). *CIF applications. VI. CIFLIB: an application program interface to CIF dictionaries and data files*. *J. Appl. Cryst.* **30**, 79–83.