International Tables for Crystallography (2006). Vol. G, Section 5.3.5.3.3, pp. 514–515.

```
_cell_length_a Ng\nobreak\cella
_cell_length_b Ng\nobreak\cellb
_cell_length_c Ng\nobreak\cellc

_publ_author_name Na\author
_publ_author_address Na\address
_publ_author_footnote NX\aufootnote
_publ_contact_author_email NX\email

_atom_site_aniso_label TU\relax
_atom_site_aniso_U_11 TU{\hfill $U^{11}$ \hfill}
_atom_site_aniso_U_12 TU{\hfill $U^{12}$ \hfill}
_atom_site_aniso_U_13 TU{\hfill $U^{13}$ \hfill}
_atom_site_aniso_U_22 TU{\hfill $U^{22}$ \hfill}
_atom_site_aniso_U_23 TU{\hfill $U^{23}$ \hfill}
_atom_site_aniso_U_33 TU{\hfill $U^{33}$ \hfill}
```

Fig. 5.3.5.7. Example map file for use with *ciftex*.

```
#[:\newif\ifproof \prooftrue
#]:\iftwocol\vfill\enddoublecolumns\fi
#a:\pretolerance1000\parskip0pt\tolerance5000
#a:\vskip10pt
#g:
#g:%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#g:\iftwocol\enddoublecolumns\twocolfalse\fi
#g:\tenbf Experimental
#g:\noindent\ninebf Compound \datablock\vskip2pt
#g:\noindent\nineit Crystal data\par
#g:\vskip2pt\begindoublecolumns\twocoltrue\defaultfont
#U:%%%%%%%%%%% Table of anisotropic U's %%%%%%
#U:\iftwocol\enddoublecolumns\twocolfalse\fi
#U:\rm Table \tableno. \it Anisotropic displacement
#U:parameters \rm (\AA$^2$) for \datablock
#U:\vskip 6pt
```

Fig. 5.3.5.8. Example format file for *ciftex*.

If the initial *character* of the line is a hash mark #, the line is treated as a comment and discarded.

5.3.5.3.1.4. *The ancillary format file*

Because a printed paper may be more verbose than its parent CIF data file, it is necessary to add text to the output from *ciftex* to represent section headings, line spaces or other formatting instructions. The program reads an ancillary file, known as the format file, for such additional text.

Each line in the format file begins with a hash mark #, a single ASCII character and a colon. The second character is chosen to match the corresponding locator character associated with data names in the map file. The rest of the line is text to be output. When the locator character associated with the data name currently being processed differs from the previous one, the output text from all lines in the format file with the new locator character are output.

The special strings `#[:` and `#]:` indicate text to be emitted at the beginning and end of the output stream, respectively.

Fig. 5.3.5.8 is an example of a simplified format file. The first line is printed at the start of the output TEX file; the second line at the end. The next line will be printed on the first occurrence of a data name flagged with the locator code `a` in the map file. In this example, that will be the name or address of an author of the paper; some typographic directives are emitted immediately before the authors' names and addresses, including the introduction of a blank line ('vertical skip', or 'vskip') of height 10 typographic points.

The lines beginning `#g:` are emitted immediately before the first data name in the group that is associated with locator code `g`. In this example, the effect is to output a heading and subheading before printing the cell-length parameters and to switch to double-column format. The line containing *only* the characters `#g:` provides for the introduction of a blank line into the TEX file, with the sole purpose of making the file more readable by human editors.

The lines beginning `#U:` are emitted at the beginning of the table of anisotropic U values.

The mechanism looks complicated at first sight, but addresses the need to generate headings at standard locations in a printed paper when the exact content of the paper is not known in advance.

The different format for directives in the map and format files means that the same file can be used for both purposes, if required. In practice it is often easier to maintain different files: the same mapping between CIF data names and TEX macros might be common to different journals, while each journal uses its own format file.

5.3.5.3.2. *Invocation of the program*

The program reads a CIF on the standard input channel and outputs TEX code on standard output. There is no provision to specify file names. It is therefore invoked within a Unix-style operating system by a command such as

```
ciftex < infile > outfile
```

where *infile* and *outfile* are the input and output files respectively; or it may be called as part of a pipeline of procedures:

```
program 1 < infile | ciftex | program 2 ...
```

A number of command-line options may be supplied to modify the operation of the program. Other than the specification of the map and format files, they are largely relevant to differing house styles for IUCr journals.

The options -map *mapfile* and -format *formatfile* specify the names of the ancillary map and format files. If not specified, they are sought in default locations on the user's file system (different values may be defined when the program is compiled) or as specified in the environment variables `$CIFTEX_MAP` and `$CIFTEX_FORMAT`, respectively.

The options *-H* and *-N* specify, respectively, whether or not hydrogen atoms in coordinate tables should be printed. The hydrogen-atom lines in the table are in fact always emitted on standard output, but in the case of the *-N* option are prefixed by a `%` (TEX comment) character and so ignored by TEX.

Options *-c* and *-F* specify the printing of centred decimal points or commas for decimal points, respectively. Finally, the option *-d* modifies certain assumptions that *ciftex* makes when typesetting CIF dictionaries. The details are of interest only to a specialist.

5.3.5.3.3. *Some general comments*

Although *ciftex* is available for public use and redistribution within the academic community, it is clearly of most interest to users who need to generate typeset representations of the contents of CIFs. Nevertheless, some elements of its design are relevant to other applications that perform on-the-fly file transformations on a strictly syntactic basis.

First, the functionality is very simple, essentially tokenizing the input data stream and exchanging tokens for replacement text as directed. An immediate consequence of this is the need for additional utilities to manipulate the input file if, for example, the data need to be presented in a particular order. In the journals production process, *QUASAR* is used to reorder an input file before passing it to *ciftex*.

Second, the replacement text should be externalized as much as possible. The use of map and format files means that the same basic program can be used for formatting according to any set of typographic rules; only the ancillary files need to be modified. In the current version of *ciftex*, the program performs some replacements internally; an objective of further development is to remove this function from the program and to externalize it either in more sophisticated table lookup files or in separate methods modules.

Third, the concept of replacement should be abstracted as much as possible. The software was written initially with the objective of replacing data names with TEX macros. Experience suggests that a generic transformation program could be written with the philosophy of replacing data names and data values by directives implemented before and after the occurrence of the data value, and as events upon its first, last and intervening occurrences. Such 'directives' and 'events' could be mapped to arbitrary replacement strings in any markup scheme, such as SGML, XML, HTML, TEX, LATEX or commercial word-processing encodings.

5.3.6. Libraries for scripting languages

Recent years have seen a great increase in the popularity of scripting languages – broadly, computer languages where the source code is run-time interpreted and that have powerful facilities for interacting with other processes and operating-system utilities. In part this is because such languages lend themselves to rapid prototyping; but the powerful features of languages such as Perl (Wall *et al.*, 2000), Python (van Rossum, 1991) and Tcl/Tk (Ousterhout, 1994) make it entirely feasible to create and maintain complex programs that can run efficiently. Several of the applications discussed in this chapter make effective use of one or more of these languages, either solely or alongside more traditional compiled languages.

As the scripting languages have become more powerful, they have also acquired more structure, so that authors now frequently build libraries (or 'modules') of functions or subroutines that can be re-used in a range of applications. This is a welcome development, for the availability of public libraries not only reduces the effort required to develop new applications, but also goes a long way towards establishing common application programming interfaces that help to standardize the way in which software from different sources is developed.

In this section two available libraries of this type are reviewed: *STAR::Parser* and *PyCifRW*.

5.3.6.1. *STAR::Parser* and related Perl modules

A collection of Perl modules has been developed at the San Diego Supercomputer Center (Bluhm, 2000) to provide basic library routines for object-oriented manipulation of STAR files with restricted syntax appropriate to CIF applications. The module name suggests that a more complete STAR implementation may be considered in future developments, but at present the modules do not handle nested loops or the inheritance of data values from global blocks. Indeed, they are still rather limited in scope; nevertheless, for the programmer wishing to prototype CIF applications in Perl, they offer a very rapid entry to parsing CIFs and constructing useful data structures that can be manipulated with standard Perl tools.

The use of some of the modules is illustrated in Fig. 5.3.6.1, which is a simplified version of the main program loop in the application written to typeset the CIF dictionaries printed in Part 4 of this volume.

5.3.6.1.1. *STAR::Parser*

STAR::Parser is the basic parsing module and may parse either data files or dictionaries (which may include save frames). It contains a single class method, *parse*, which returns an array of DataBlock objects. Each DataBlock object contains all the data items within an individual data block of the file. Even if the file contains only a single data block, the resulting object is passed in an array.

The contents of the data blocks may be accessed and manipulated by the methods provided by the *STAR::DataBlock* and *STAR::Dictionary* modules. They are stored internally as a multidimensional hash (the Perl term for an associative array with keys and associated values, which may themselves be complex data objects). Keys are provided for data blocks, save blocks, categories and data items identified during the parse. The module provides no error checking of files or objects, against the dictionary or otherwise – limited checking functionality is available through other modules in this collection.

In the example of Fig. 5.3.6.1, the *parse* method is called at line 9 to read a DDL2 dictionary file (indicated by the `-dict=>1` parameter) and return an array of data blocks. In DDL2 dictionaries (such as the mmCIF dictionary of Chapter 4.5) an entire dictionary is contained within a data block; save frames partition the data block into definitions for separate items. Normally a DDL2 CIF dictionary has only a single data block; nevertheless, the example program can handle multiple data blocks in the array, and traverses the one or several data blocks in the array through a Perl *foreach* construct (line 15).

5.3.6.1.2. *STAR::Dictionary*

The *STAR::Dictionary* module contains class and object methods for Dictionary objects created by the *STAR::Parser* module, and is in fact a subclass of *STAR::DataBlock* (see next section). Since CIF dictionaries are fully compliant STAR files, they require little that is different from the methods developed for handling data files. The method *get_save_blocks* is provided to return an array of all save frames found in the Dictionary object.

In line 25 of the example, the method is called on each dictionary loaded from the input file (as described above, normally there will only be one). The method is combined with the Perl *sort* function to create an array of save frames from the dictionary, arranged in alphabetic order. All further manipulations of the contents of these save frames will use the methods of the generic *STAR::DataBlock* class.

5.3.6.1.3. *STAR::DataBlock*

This package provides several useful methods for handling the objects within a data block returned by the *STAR::Parser* module.

The class has a constructor method *new*, which can create a completely new DataBlock object if called with no argument. This is of course essential for applications that wish to write new CIFs. Alternatively, it may be called with a `$file` argument to retrieve an existing object that has previously been written to the file system using the *store* object method described below:

```
$data_obj = STAR::DataBlock->new{ -file=>$file };
```

Table 5.3.6.1 summarizes the object methods provided by the package. The *store* method allows a DataBlock object to be serialized and written to hard disk for long-term storage. The Perl public *Storable::* module is used.

The *get_item_data* method returns the data values for a named data item. It is used frequently in the example program of Fig. 5.3.6.1; for example, at lines 28–30 the array of categories

references