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

```
 1 #!/usr/local/bin/perl
 2
 3 use STAR::Parser;
 4 use STAR::DataBlock;
 5 use STAR::Dictionary;
 6
 7 my $file = $ARGV[0];
 8
 9 @dicts = STAR::Parser->parse(-file=>$file, -dict=>1);
10
11 exit unless ( $#dicts >= 0 ); # exit if nothing there
12
13 print_document_header;
14
15 foreach $dictionary (@dicts) { # usually one
16   # Get the dictionary name and version
17   $dictname = ($dictionary->get_item_data(
18                            -item=>"_dictionary.title"))[0];
19   $dictversion = ($dictionary->get_item_data(
20                          -item=>"_dictionary.version"))[0];
21
22   print "Dictionary $dictname, version $dictversion\n";
23
24   # Get list of save-frame names, sorted alphabetically
25   @saveblocks = sort $dictionary->get_save_blocks;
26
27   foreach $saveblock (@saveblocks) { # For each save frame
28     @categories = $dictionary->get_item_data(
29                                    -save=>$saveblock,
30                                    -item=>"_category.id");
31     if ( $#categories == 0 ) { # category save frame
32       format_category($saveblock); # format the category
33
34       $category = $categories[0] ;
35       foreach $block (@saveblocks) { # re-traverse blocks
36         @itemcategoryids = $dictionary->get_item_data(
37                                    -save=>$block,
38                                -item=>"_item.category_id");
39         @itemname = $dictionary->get_item_data(
40                                    -save=>$block,
41                                    -item=>"_item.name");
42
43         if ( $#itemcategoryids == 0) { # category pointer
44           format_item($block)
45                if $itemcategoryids[0] =  /^$category$/i;
46         } elsif ( $#itemname == 0 &&
47                     $itemname[0] =  /^_$category\./i ) {
48           format_item($block);
49         }
50
51       } # end foreach of items matching current category
52     }
53   } # end foreach of save frames in the dictionary
54 } # end foreach loop per included dictionary
55
56 print_document_footer;
57 exit;
```

Fig. 5.3.6.1. Skeleton version of an application to format a CIF dictionary for publication. Only the main program fragment is shown. Line numbering is provided for referencing in the text. *format_category*, *format_item* and the *print_document_** commands are calls to external subroutines not included in this extract.

in the input dictionary is assembled by retrieving the value associated with the data item **_category.id** as the component save frames are scanned in sequence. Note that for applications within dictionaries, the method takes a `-save=>$saveblock` parameter to allow the extraction of items from specific save frames. For manipulations of data files, this parameter is omitted. In lines 17–20, the method is called without this parameter because the name and version of the dictionary are expected to be found in the outer part of the file, not within any save frame.

get_keys allows a user to display the structure of a CIF or dictionary file and can be used to analyse the content of an unknown input file. When written to a terminal, the string that is returned by this method appears as a tabulation of the items present at the different levels in the data structure hierarchy, each level in the hierarchy being indicated by the amount of indentation (Fig. 5.3.6.2).

Table 5.3.6.1. *Object methods provided by the STAR::DataBlock Perl module*

Method	Description
store	Saves a DataBlock object to disk
get_item_data	Returns all the data for a specified item
get_keys	Returns a string with a hierarchically formatted list of hash keys (data blocks, save blocks, categories and items) found in the data structure of the DataBlock object
get_items	Returns an array with all the items present in the DataBlock
get_categories	Returns an array with all the categories present in the DataBlock
insert_category	Inserts a category into a data structure
insert_item	Insert an item into a data structure
set_item_data	Sets the data content of an item according to a supplied array

The *get_items* and *get_categories* methods are largely self-explanatory. The items or categories in the currently active DataBlock object are returned in array context.

insert_item and *insert_category* are the complements of these methods, designed to allow the insertion of new items or categories. Where appropriate (*i.e.* in dictionary applications), the save frame into which the insertion is to be made can be specified.

The remaining method, *set_item_data*, is called to set the data of item `$item` to an array of data referenced by `$dataref`:

```
$data_obj->set_item_data( -item=>$item,
                          -dataref=>$dataref );
```

As usual, an optional parameter `-save=>$save` may be included for dictionary applications where a save frame needs to be identified; the value of the variable *$save* is the save-frame name.

Note that the current version of the module does not support the creation and manipulation of data loops, although the *get_item_data* method will correctly retrieve arrays of data values from a looped list.

There are five methods available to set or retrieve attributes of a DataBlock object, namely: *file_name* for the name of the file in which the DataBlock object was found; *title* for the title of the DataBlock object (*i.e.* the name of the CIF data block with the leading **data_** string omitted); *type* for the type of data contained – 'data' for a DataBlock object but 'dictionary' for an object in the STAR::Dictionary subclass; and *starting_line* and *ending_line* for the start and end line numbers in the file where the data block is located. The method *get_attributes* returns a string containing a descriptive list of attributes of the DataBlock object.

5.3.6.1.4. *STAR::Checker*

This module implements a set of checks on a data block against a dictionary object and returns a value of '1' if the check was successful, '0' otherwise. The check tests a specific set of criteria:

(i) Are all items in the DataBlock object defined in the dictionary?

(ii) Are mandatory items present in the data block?

(iii) Are dependent items present in the data block?

(iv) Are parent items present?

(v) Do the item values conform to item type definitions in the dictionary?

Obviously, these criteria will not be appropriate for all purposes, and are in any case fully developed only for DDL2 dictionaries. An optional parameter `-options=>'1'` may be set to write a list of specific problems to the standard error output channel.

```
 data  save
block  block  categ. item
---------------------------

cif_img.dic
        -
                _category_group_list
                        _category_group_list.description
                        _category_group_list.id
                        _category_group_list.parent_id
                _dictionary
                        _dictionary.datablock_id
                        _dictionary.title
                        _dictionary.version
                _dictionary_history
                        _dictionary_history.revision
                        _dictionary_history.update
                        _dictionary_history.version
                _item_type_list
                        _item_type_list.code
                        _item_type_list.construct
                        _item_type_list.detail
                        _item_type_list.primitive_code
                _item_units_conversion
                        _item_units_conversion.factor
                        _item_units_conversion.from_code
                        _item_units_conversion.operator
                        _item_units_conversion.to_code
                _item_units_list
                        _item_units_list.code
                        _item_units_list.detail
        ARRAY_DATA
                _category
                        _category.description
                        _category.id
                        _category.mandatory_code
                _category_examples
                        _category_examples.case
                        _category_examples.detail
                _category_group
                        _category_group.id
                _category_key
                        _category_key.name
```

Fig. 5.3.6.2. Structure of the imgCIF dictionary (Chapter 4.6) as described by the *get_keys* method of the *STAR::DataBlock* module. Only the high-order file structure and the contents of the first category are included in this extract.

5.3.6.1.5. *STAR::Writer and STAR::Filter*

Two other modules are supplied by this package. *STAR::Writer* is a prototype module that can write STAR::DataBlock objects out as files in different formats; currently only the *write_cif* method exists to output a conformant CIF. *STAR::Filter* is an interactive module that prompts the user to select or reject individual categories from a STAR::Dictionary object when building a subset of the larger dictionary.

5.3.6.2. *PyCifRW*: CIF reading and writing in Python

PyCifRW (Hester, 2006) is a simple CIF input/output utility written in Python. It does not validate content against dictionaries, but it does provide a robust parser that has been extensively tested against various test files containing subtle syntactic features and against the collection of over 18 000 macromolecular CIFs available from the Protein Data Bank. The parser was implemented using the *Yapps2* parser generator (Patel, 2002) and is based on the draft Backus–Naur form (BNF) developed during a community exercise to review the CIF specification of Chapter 2.2.

As with the Perl library discussed above, *PyCifRW* presents an object-oriented set of classes and methods. Two classes are provided, *CifFile* and *CifBlock*.

A *CifFile* object provides an associative array of *CifBlock* objects, accessed by data-block name.

The methods available for the *CifFile* type are: *ReadCif(filename)*, which initializes or reinitializes a file to contain the CIF contents; *GetBlocks()*, which returns a list of the data-block names in the file; *NewBlock(blockname, [block contents])*, which adds a new data block to the file object; and *WriteOut(comment)*, which returns the contents of the current file as a CIF-conformant string, with an optional comment at the beginning. For the *NewBlock* method, the optional *block contents* must be a valid *CifBlock* object, as described below. The *NewBlock* method returns the name of the new block created, which will *not* be the requested *blockname* if a data block of the same name already exists (this conforms to the STAR and CIF requirement that data-block names must be unique within a file).

A *CifBlock* object represents the contents of a data block. The methods available to retrieve or manipulate the contents are: *GetCifItem(itemname)*, which will return the value of the data item with data name given by *itemname* (and which can be a single value or an array of looped values); *AddCifItem(data)*, which adds *data* to the current block, where *data* represents either a data name and an associated single value, or, for the case of looped data, a tuple containing an array of data names and an array of arrays of associated data values; *RemoveCifItem(dataname)*, to remove the specified data item from the current block; and *GetLoop(dataname)*, which returns a list of all data items occurring in the loop containing the data name provided. If *dataname* does not represent a looped data item, an error is returned.

The *GetLoop* method is important for the proper handling of looped data, and care is taken to handle loops robustly and efficiently. Items that are initially looped together are kept in the same loop structure.

Both *CifFile* and *CifBlock* objects act as Python mapping objects, which has the advantage that the value of a data item can be read or changed using an intuitive square-bracket notation. For example, a program can retrieve the value of a data item named `_my_data_item_name` in block 'myblockname' of a previously opened file *cf* using the following syntax:

```
value = cf["myblockname"]["_my_data_item_name"]
```

The returned value is either a single item, or a Python array of values if the data item occurs in a loop. Values are set in an analogous way and other common operations with mapping objects are also implemented.

5.3.7. Rapid development tools

The programs described so far in this chapter tend to fulfil a single purpose. Each program addresses a single clearly defined task and is of benefit to a user who has no need or desire to write a customized program. Where there is a need to write a new application, libraries of subroutines are available to the full-time programmer, such as those described in Chapters 5.4 to 5.6. However, in between these extremes, there are a large number of cases for which there is a need to combine the functionality of a number of existing programs without incurring the overhead of writing a new integrated application.

This section describes utilities that assist the development of new applications from existing software.