A lexical analyser for STAR/CIF/mmCIF data

Peter A. Keller, Global Phasing Ltd., Cambridge, UK.
13 September 2013

Some of the subtleties involved in handling the low-level syntax of STAR/CIF/mmCIF data are discussed. A small programmers' library is introduced which converts such data to a sequence of tokens identified by type, and allows an application programmer to write CIF parsers without needing to be concerned with those lexical subtleties. The library is written in Java, but could be readily ported to C/C++, Python or Perl.

Introduction

The mmCIF format[1] has for some years been the underlying standard for the PDB archive[2], but is also becoming more widely used by working crystallographers to exchange data between applications from different authors. In particular, the use of chemical dictionaries in mmCIF format by refinement and visualisation programs has become very widespread. In practice though, crystallographers still encounter interoperability problems when working with mmCIF-format data using applications written by different authors. There are a number of reasons for this (which are currently being addressed by the mmCIF working group[3]) but the focus here is on the lowest level: the basic STAR syntax[4] (a subset of which is used for CIF[5] and mmCIF).

While the STAR syntax appears straightforward when examining examples of CIF or mmCIF data, the details of the full specification are complex and hard to understand, and several features are different to those found in other textual formats that are familiar to scientific software developers. These features create "corner cases" where different STAR/CIF/mmCIF parsers differ in their interpretation of the data. Crystallographers experience these differences as application failures during their day-to-day work.

An example of a STAR corner case

Consider the following fragment of a chemical information dictionary in mmCIF format (from an example downloaded from the PDBe[6]):

```
loop_
_chem_comp_atom.comp_id
_chem_comp_atom.atom_id
_chem_comp_atom.alt_atom_id
_chem_comp_atom.type_symbol
_chem_comp_atom.charge
_chem_comp_atom.pdbx_align
_chem_comp_atom.pdbx_aromatic_flag
_chem_comp_atom.pdbx_leaving_atom_flag
_chem_comp_atom.pdbx_stereo_config
_chem_comp_atom.model_Cartn_x
_chem_comp_atom.model_Cartn_y
_chem_comp_atom.model_Cartn_z
_chem_comp_atom.pdbx_model_Cartn_x_ideal
_chem_comp_atom.pdbx_model_Cartn_y_ideal
_chem_comp_atom.pdbx_model_Cartn_z_ideal
_chem_comp_atom.pdbx_component_atom_id
_chem_comp_atom.pdbx_component_comp_id
_chem_comp_atom.pdbx_ordinal
      ...
PGP "O5'"  O5*  O 0 1 N N N -3.724 45.688 -51.277 -1.973 -0.831 2.790  "O5'"  PGP 9
```

This seems straightforward enough: the atom name `O5'` contains an apostrophe character, so it has been double-quoted in order to remove any ambiguity about whether or not the apostrophe is part of the data value. However, consider the case where such an atom name is not quoted, so the line of data values looks like this:

```
PGP O5'  O5*  O 0 1 N N N -3.724 45.688 -51.277 -1.973 -0.831 2.790  O5'  PGP 9
```

How should an application treat such data? For a developer unfamiliar with the details of the STAR or CIF specifications, various possibilities might seem to be available:

- Treat it as a syntax error: it is "obvious" that you can't have a text delimiter appearing in an unquoted text value without some form of escaping
- Be strict about using apostrophes as delimiters, i.e. treat the two apostrophes as a delimiting pair so that the data contains this single-quoted text token:
  ```
  ' O5* O 0 1 N N N -3.724 45.688 -51.277 -1.973 -0.831 2.790 O5'
  ```
  The parser will probably fail in this case, because the number of data values in the whole loop is unlikely to be an exact multiple of the number of data names in the loop header. Even if that doesn't happen, the application will almost certainly fail at a later stage when given this value as a `_chem_comp_atom.alt_atom_id`
- Treat `O5'` as a valid non-quoted, space-delimited text token and accept it as the data value without any errors.

The correct answer according to the STAR/CIF specifications is in fact the third one: a `'` or `"` character can appear in an unquoted string in STAR, providing that it is not the first character (see below). However, different libraries/applications have been observed to treat this in different ways. Users experience such different treatments as breakage when attempting to use the same files with different applications. BUSTER[7] uses the RCSB CIFPARSE-OBJ[8] library for some of its handling of mmCIF data, and we found when investigating an mmCIF-related problem on behalf of a BUSTER user that the CIFPARSE-OBJ library and COOT[9] differed in their handling of this type of non-quoted string. The version of CIFPARSE-OBJ that was current at the time treated it as a syntax error whereas COOT correctly accepts it as a valid data value. (CIFPARSE-OBJ has since been updated and handles this data correctly from version 7.105.)

Handling single and double quotes

The rules for single and double quotes are complex, but may be summarised as follows: they are only treated as delimiters for quoted text if it "makes sense" to treat them that way. Otherwise, they are treated as ordinary characters in string values. This is discussed in the CIF specification[10] section 2.2.7.1.4 paragraph (15), where the following is given as an example of a valid data item:

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

Here the second apostrophe is not treated as a closing delimiter: it would need to be followed by whitespace to have that role. The part of the specification that states this formally is in section A2.1.1.2 of the STAR specification[4], in the following production for the string that appears within the delimiting single quotes:

```
<S_quote_string> ::= { '\'' <non_blank_char> | <not_an_S_quote }* { '\'' }*
```

In other words, a single-quoted string may contain a single quote followed by a non-whitespace character (incuding another single quote). It may also end with any number of single quote characters in addition to the single quote followed by whitespace that acts as the closing delimiter of the string. Section (e) of Table 2.2.7.1 in the CIF specification makes an equivalent statement in a slightly different way.

The formal specification of non-quoted strings such as the `O5'` example above are complicated slightly by the requirement to treat the semicolon character (`;`) in two different ways depending on whether it occurs at the start of a line or not. However the productions in section A2.1.1.2 of [4] for non-quoted strings both have the following term after the specification of the first character:

```
<non_blank_char>*
```

A "non_blank_char" is defined as either an "ordinary_char" or one of the following characters: `" # $ ' ; _ [ ]` from which it is clear that all of these characters may appear in a non-quoted text string as long as they are not the first character of that text string.

A final point to note is that there is no mechanism for escaping the three text delimiter characters used in CIF data so that they are not treated as opening/closing delimiters when preceded/followed by whitespace. This means that a string such as the following (which contains both single and double quotes followed by whitespace):

```
The atom name is O5' which is the "new" convention
```

can only be represented in STAR-format data as a semicolon-delimited text string as follows:

```
;The atom name is O5' which is the "new" convention
;
```

It cannot be quoted using single or double quotes. The same consideration applies to `<newline>;` in semicolon-delimited text, which cannot contain the semicolon character at the start of a line as part of its value.

Privileged constructs and the STAR format

It is obvious that in CIF and mmCIF data, the string `loop_` and strings that start with one of `_ # ;` (at the start of a line) or `data_` have a specific role in the syntax of the file. In order to avoid data values clashing with these constructs, they must be quoted, for example:

```
_example "data_value"
```

The STAR syntax specifies several similar constructs that are not part of the CIF/mmCIF standards. However, in order to ensure that CIF/mmCIF data conforms to the STAR syntax, these constructs must also be quoted when they appear as data values. Applications that read CIF-format data should treat all unquoted occurences of these constructs as syntax errors. The constructs are:

- The strings `stop_` `global_` and `save_`
- Strings starting with `save_` `[` `]` or `$`

The two versions of the STAR syntax specification in [4] and section 2.2.3 of [10] conflict with each other over whether strings that start with `stop_` `global_` or `loop_` but have at least one additional non-whitespace character can be treated as non-quoted text strings. To avoid ambiguity, it therefore seems best to quote such values, e.g. this:

```
_example "global_value"
```

rather than this:

```
_example global_value
```

The GPhL StarTools tokeniser

The GPhL StarTools tokeniser takes STAR-format data as input, breaks it down into "tokens" (each of which corresponds to a data name, a data value, or another STAR construct), and returns a sequence of those tokens. Each returned token is assigned a type. An application programmer can use this sequence of tokens to populate data structures of their own choosing without needing to be concerned with low-level lexical subtleties such as those described above. Once the tokeniser has been instantiated and provided with input, the sequence of tokens can be retrieved in the familiar style of classes such as java.io.StreamTokenizer:

```
StarToken tok;
while ( tokeniser.hasMoreTokens() ) {
    tok = tokeniser.nextToken();
    ... code to handle returned token ...
}
```

The `StarToken` class provides methods `getValue()` and `getType()` which will return the value and type respectively of the token. The value will have had quoting removed, but the token type will still identify the type of quoting that was used in the input data.

At the heart of the tokeniser is a long regular expression that matches STAR data. Individual capturing groups in the regular expression correspond to specific token types, including tokens that break STAR syntax rules. Also, two token types are defined for unquoted `.` and `?` characters: these are not special tokens in the STAR syntax, but in CIF they are specified to mean "not applicable" and "unknown" respectively. No error handling is provided by the tokeniser: this is left up to the calling application or parser, but the tokeniser provides information to help with this.

The regular expression encapsulates most of the logic and decision making used in breaking down the input data into tokens, and as a consequence it uses some advanced features of regular expressions such as non-capturing groups, lazy quantifiers and zero-width lookahead and lookbehind assertions. The Java code proper in the tokeniser is very simple, and as a consequence porting the code to another language that supports the same advanced regular expression features should be straightforward. In particular, Perl and Python should be able to use the same regular expression directly since their built-in regular expression languages are similar to that of Java. C and C++ will require the use of an advanced regular expression library such as the Perl Compatible Regular Expressions library[11] rather than the POSIX-style regular expression libraries that are often provided with C/C++ compilers.

The StarTools tokeniser may be downloaded from http://www.globalphasing.com/startools/startools-0.2.0-full.tar.gz (the download also contains a simple working example program), and the Javadoc can be viewed at http://www.globalphasing.com/startools/javadoc/. It may be freely used in both open and closed source libraries and applications, under the terms of a BSD-like licence.

References

1: macromolecular Crystallographic Information File, http://mmcif.rcsb.org/
2: Worldwide Protein Data Bank, http://www.wwpdb.org/index.html
3: PDBx/mmCIF Working Group, http://www.wwpdb.org/workshop/wgroup.html
4: S.R. Hall and N. Spadaccini, Specification of the STAR file, International Tables for Crystallography (2006) . Vol. G, ch. 2.1, pp. 13-19. http://dx.doi.org/10.1107/97809553602060000727
5: IUCr CIF resources, http://www.iucr.org/resources/cif
6: Protein Data Bank in Europe, http://www.ebi.ac.uk/pdbe/
7: BUSTER, http://www.globalphasing.com/buster/
8: CIFPARSE-OBJ C++ Language Class Library of mmCIF Access Tools, http://sw-tools.pdb.org/apps/CIFPARSE-OBJ/index.html
9: P. Emsley, B. Lohkamp, W.G. Scott, K. Cowtan, Features and Development of Coot, Acta Crystallographica (2010). D66, 486-501. http://www2.mrc-lmb.cam.ac.uk/personal/pemsley/coot/
10: S. R. Hall, J. D. Westbrook, N. Spadaccini, I. D. Brown, H. J. Bernstein and B. McMahon, Specification of the Crystallographic Information File (CIF), International Tables for Crystallography (2006) . Vol. G, ch. 2.2, pp. 20-36. http://dx.doi.org/10.1107/97809553602060000728
11: Perl Compatible Regular Expressions library, http://www.pcre.org/

This article is available in PDF form at http://www.globalphasing.com/startools/StarTools_article.pdf