Parser Combinators: a Practical Application for Generating Parsers for NMR Data

Matthew Fenwick[1], **Gerard Weatherby**[2], **Heidi JC Ellis**[2], and **Michael R. Gryk**[1]

[1]Department of Microbial, Molecular and Structural Biology, University of Connecticut Health Center, 263 Farmington Avenue Farmington, Connecticut 06030

[2]Department of Computer Science / Information Technology, Western New England University, Springfield, Massachusetts

Abstract

Nuclear Magnetic Resonance (NMR) spectroscopy is a technique for acquiring protein data at atomic resolution and determining the three-dimensional structure of large protein molecules. A typical structure determination process results in the deposition of a large data sets to the BMRB (Bio-Magnetic Resonance Data Bank). This data is stored and shared in a file format called NMR-Star. This format is syntactically and semantically complex making it challenging to parse. Nevertheless, parsing these files is crucial to applying the vast amounts of biological information stored in NMR-Star files, allowing researchers to harness the results of previous studies to direct and validate future work. One powerful approach for parsing files is to apply a Backus-Naur Form (BNF) grammar, which is a high-level model of a file format. Translation of the grammatical model to an executable parser may be automatically accomplished. This paper will show how we applied a model BNF grammar of the NMR-Star format to create a free, open-source parser, using a method that originated in the functional programming world known as “parser combinators”. This paper demonstrates the effectiveness of a principled approach to file specification and parsing. This paper also builds upon our previous work [1], in that 1) it applies concepts from Functional Programming (which is relevant even though the implementation language, Java, is more mainstream than Functional Programming), and 2) all work and accomplishments from this project will be made available under standard open source licenses to provide the community with the opportunity to learn from our techniques and methods.

Keywords

component; functional-programming; Java; NMR; parsing; NMR-Star

I. Introduction and Background

A. NMR background

In the last 30 years, Nuclear Magnetic Resonance (NMR) has emerged as an important technique for obtaining atomic-resolution data on proteins in the areas of macromolecular structure, dynamics, protein chemistry and intermolecular binding. Such data are of high importance in the field of drug design: knowing the structure and chemistry of protein active sites is important for designing molecular inhibitors. Two major challenges still facing NMR protein spectroscopy are obtaining high-quality data from relatively large proteins -- that is, proteins larger than about 30 kilodaltons -- and efficiently and correctly processing and interpreting the vast amounts of data produced in the course of the structure determination process.

B. An overview of the NMR structure determination process

Protein structure determination is a multi-step process. First, a researcher must prepare a high quality protein sample. Second, the protein sample is inserted into an NMR spectrometer, where a series of experiments are performed, producing time-domain data containing sinusoidal relaxations. Third, this data is computationally transformed into frequency-domain spectra, using a series of mathematical operations such as Fourier transforms and windowing. Fourth, the spectra are pick-peaked, a process that identifies the signals in the spectra as peaks. Fifth, the peaks are correlated with atoms in the protein in a process called “chemical shift assignment”. Sixth, a different variety of NMR data, known as NOESY data, is assigned based on the chemical shift assignments; NOESY data indicates through-space interactions between spatially adjacent hydrogen atoms. Seventh, after many NOESY interactions have been identified, a structure calculation program finds a protein conformation(s) that satisfies the NOESY restraints.

C. NMR and computation

NMR studies typically involve the generation and processing of huge amounts of data, which are stored in flat files in a large variety of file formats. Data collected from the spectrometer are stored in binary format with file sizes ranging from megabytes to gigabytes, depending on the type of experiment being run. Subsequent data analysis and interpretation proceeds through a series of phases, during which the data is successively transformed from time-domain binary data to frequency-domain binary data, then to peak lists, chemical shift assignments, NOESY assignments, and atomic coordinates, all in various ASCII formats. [2] A subset of this data is deposited in the public database, BioMagResBank (BMRB) [3], where it is converted to the NMR-Star format.

D. Functional Programming

Functional Programming, or FP for short, has long been touted as an important field of programming study and application, due to its facilities for composition and abstraction, which allow it to provide extremely high-level, abstract, concise, and correct solutions to difficult problems [4].

The high-level nature of FP has also proven to be a fertile hotbed for a large number of powerful, declarative algorithms and techniques. Often such algorithms and techniques, after being initially developed and applied in an FP setting, are found to be extremely useful and practical and are then borrowed and applied in mainstream languages. The result is that mainstream languages continually add features from FP which are found to be useful. One important example is garbage collection, which was originally invented for Lisp, an early FP language [5].

E. Grammars, BNF, and their application to parsing and compilers

Backus-Naur Form (BNF)[6] is a notation for context free grammars. It is used to provide a specification of the syntax of both programming languages and of file formats. The chief advantages of BNF grammars are that they are formal, declarative, and simple -- a few lines can describe a complex language. BNF grammars are provided for most popular programming languages in use today; many tools exist which are able to convert a grammar into an executable parser for the language being described. This has given rise to tools such as YACC (Yet Another Compiler Compiler).

F. Parsing combinators

Parser combinators [7] are a means of implementing grammar-based parsers entirely within a single programming language (compare to tools such as Lex/Yacc, which require that a

programmer use outside, separate tools). The parsers benefit from such a complete integration in that they are able to directly use and be used by any methods/objects available in the host language, in areas such as:

- Testability: the parsers can be tested using the standard unit testing library of the programming language in which they are written.
- Expressability: the parsers can express any constraint, condition, or transformation expressible in the host language.
- Composability: parsers can be written separately, then combined in arbitrary ways. Note that this synergizes with testability, as parsers can be tested and verified in isolation before being combined.
- Flexibility: parsers can be used as typical data structures within the host language, whether for serialization, BNF grammar generation, runtime debugging.
- Maintainability: parsers in a host language are maintained using the same methods as all other code in that language is maintained. Additionally, the lack of a separate tool reduces the barrier to understanding for future developers, who will not be tasked with learning how to use an additional tool, or with integrating it into the development process or deployed product.
- Learnability: parser combinators are provided as a library written in the development language of choice; thus, learning to use them can be as simple as browsing the Application Programming Interface (API) documentation, using an Integrated Development Environment (IDE) such as Eclipse.

Importantly, BNF grammars can also be constructed within a program using parser combinators. Furthermore, the basic operators used in BNF can be extended and augmented by capturing patterns in additional parser combinators. This is a major advantage of parser combinators over standard BNF grammars: the combinators are extensible, in that additional ones can be defined at any time within the program. Such an extension is not possible in standard BNF, and results in grammars that are longer, less clear, and more error-prone.

G. NMR Star files

The BMRB stores and shares its data using files in the NMR Star format [8]. Typically, each protein for which data was deposited will be represented by a Star file containing all of that information. Information that is typically present includes information about the depositors, information about the protein, information about experiments performed on the protein, and information about the interpretation of the data from those experiments.

This data can be applied to further NMR studies in a multitude of additional ways, such as:

- using the data from a structure determination as the starting point for a dynamics or binding study
- using the data from one protein as a starting point for studying similar proteins or homologues
- using the data as a test bed to assist in the develop of new computational tools
- using a large subset or the entirety of the data to conduct large-scale bioinformatics studies (data mining), investigating systematic tendencies and trends
- using the data to validate further studies by providing a comparison of tested, verified data

However, the NMR Star format is complicated enough that it is difficult and time-consuming to hand-write parsers that are correct, clear, succinct and maintainable.

H. The NMR Star format

Fig. 1 shows a section from an NMR Star file. Our description of the format consists of: 1) tokens, such as identifiers, comments, whitespace, values, keywords, and 2) syntactic structures such as loops, save frames, and key-value pairs.

I. Existing NMR Star parsers

There are several NMR Star parsers already in existence. The sans package [9] includes a lexical scanner generated using the jFlex tool and a handwritten, recursive-descent style parser. The starlibj package [10] includes a lexical scanner and a parser generated using the javacc tool. The Wattos package [11] includes an NMR-Star parser with a lexer generated using the jFlex tool and a handwritten, recursive-descent style parser.

The key differences from the parser we present are due to the choice of technology: while our parser is built using parser combinators, existing parsers were built using code generators, hand-written, or a combination of the techniques.

As is well known in the computer science community, code generators have several drawbacks: 1) continual access to and knowledge of a third-party tool, as the code is maintained and updated; as such tools are necessarily complex, their syntax may be non-trivial to learn. 2) the parser generator tools are not integrated with Java. This makes it difficult to update the parser definition, automatically build/test the code, and prevents the parser from taking advantage of functions in Java (and vice versa).

Hand-written parsers also have drawbacks: such parsers of syntactically complex languages are typically more complicated and less composable than parsers built using the parser combinator approach.

The major advantages of parser combinators, and thus of our approach, is that no separate tools are required, a single language is used to implement the parser, complex grammars are dealt with cleanly and result in easily composable units, and the correspondence between specification and implementation is clear.

II. Results

A. Publicly accessible project

All results of this project, including code, documentation, supporting material can be found on our Connjur website at http://www.connjur.org.

As we are an open-source software group, we have released all material under the BSD/MIT open source licenses and encourage interested parties to download the code and apply it to their own projects as they see fit.

B. Implementation

We were able to create an executable application which can read in a Star file, parse it, and output the results in a standard data-interchange format, JavaScript Object Notation (JSON). The implementation is similar to that of many compilers and interpreters [12], in that the conceptual stages are:

1. the input is tokenized, or broken into a sequence of tokens, which are the terminal elements of a format specification

2. the tokens are analyzed to form syntactic structures, resulting in an abstract syntax tree (AST)
3. since the NMR Star format is not a programming language, there are no static semantics (i.e. types) to check, and this step is omitted
4. the AST is then transformed into the output data (this is the equivalent of the code generation/interpretation steps of compilers/interpreters)

Steps 1-3 can be seen as the analogous to the responsibilities of a compiler's front-end; step 4 would typically be handled by a compiler's back-end.

Following this general architecture, our parser is designed in the following layers:

1. application layer: this layer is responsible for dealing with user input, the file system, and output. It locates files, reads them into memory, and sends the contents to the following layers for parsing analysis. Then it is responsible for formatting and printing the result, or reporting errors if any occur during this procedure.
2. lexical analysis layer. This layer is responsible for breaking the input string into a stream of tokens. Examples of tokens are comments, identifiers, and punctuation such as 'stop_' and 'loop_'. The token definitions were chosen so as to overlap as little as possible, but this was not entirely possible due to the constraints of the Star format. Nevertheless, we attempted to ensure that ambiguous tokenization can never occur. If the input cannot be fully lexed in this phase, an error will be reported and processing aborted.
3. syntactic analysis layer. This layer is responsible for assembling the token stream into an abstract syntax tree. Presumably, unnecessary tokens including whitespace, newlines, and comments have already been discarded before the token stream is passed here, which allows this layer to be much simpler. Similarly as in layer 2, all errors here are fatal and cause immediate abortion of the processing.

Our implementation raises a few questions. 1) Why did we choose to split parsing into two layers? It is not strictly necessary to do so; however, we had several compelling reasons to do so, which were the same reasons that many existing parsers and compilers do. First, it is more efficient to lex and parse in totally separate steps, and as Star files can get quite large, this can shorten total parsing time from several minutes to several seconds. Second, such separation helps keep the grammar simpler overall; when the grammar is simpler, it is easier to maintain and easier to find and remove defects. Third, the separation promotes meaningful and understandable error messages in the event that a parsing error does occur; it does this by localizing the context and reducing the number of alternatives that may be expected at any given stage of parsing. Fourth, the separation also allows a clear divide between abstract parsing and concrete parsing, in the sense that tokenization is solely concerned with the exact textual form of the file, while syntactic parsing is focused on how non-terminals are assembled to form grammatical structures. Again, this is another separation that keeps the grammar specification clearer, simpler and more maintainable.

C. Grammar: BNF-like specification

To develop this grammar, we started by reviewing the existing literature for Star files. While this was helpful, and we were able to uncover the specifications for older flavors of Star format, the currently used NMR Star format was not covered in such publications [13]. Thus, to figure out the exact format, we built a grammar based on the published ones, collected many examples to use as test cases for our parser, and talked to the helpful scientists and programmers at the BMRB, who were able to provide us with numerous indications and explanations of the NMR Star format's deviation from standard Star format.

This allowed us to correct the specification in cases where it was wrong, augment it where it was missing details, and formalize it where it was in prose, ambiguous or unclear.

Figures 3 and 4 show the grammar. Note that token names start with a capital letter, while non-tokens begin with a lowercase letter.

D. Java parser combinators

A library of general parser combinators, suitable for developing format-specific parsers following a similar process was also created and implemented in Java. The definitions for the basic combinators were based on those given in [14], which were typically in a dialect of ML (often Haskell). Translating the definitions into Java presented a number of difficulties which we were able to overcome, including lack of direct language support for:

- first-class functions
- algebraic datatypes
- monads
- type abbreviations

Fortunately, none of these problems was insurmountable; we were able to build on the work of others who had previously applied Functional Programming techniques within Java, including FunctionalJava [15] and Google Guava [16].

III. Methods

This software was developed using the Eclipse Integrated Development Environment in the Java programming language. We also used Concurrent Versions Systems for source control management and Maven for build configuration.

We obtained example NMR Star files from the BMRB website at http://bmrb.wisc.edu/.

IV. Future Work

Applying grammars to file parsing lays the groundwork for a robust, maintainable, and effective software system for dealing with NMR files. Such a system does not currently exist, as existing implementations are either incomplete or not based on such rigorous, high-level format specifications and are thus extremely difficult to maintain in the ever-changing world of NMR data formats.

V. Conclusions

BNF grammars are an effective method for NMR syntax description. The advantages of such an approach have long been known to the computer science field, and include their unambiguous nature and machine readability. Not only are such parsers easier to build, but also are easier, in the long run, to debug, correct, and maintain.

Combining such high level specifications with the power of a general purpose programming language, in the form of parser combinators,

Acknowledgments

This research was funded by US National Institutes of Health grant GM-083072.

References

1. Matthew, Fenwick; Colbert, Sesanker; Schiller, Martin R.; Ellis, Heidi JC.; Lee Hinman, M.; Vyas, Jay; Gryk, Michael R. An Open-Source Sandbox for Increasing the Accessibility of Functional Programming to the Bioinformatics and Scientific Communities. Ninth International Conference on Information Technology - New Generations. 2012:89–94.
2. Ellis, Heidi JC.; Nowling, Ronald J.; Vyas, Jay; Martyn, Timothy O.; Gryk, Michael R. Iterative Development of an Application to Support Nuclear Magnetic Resonance Data Analysis of Proteins. Eighth International Conference on Information Technology: New Generations. 2011:1014–1020.
3. Ulrich, Eldon L., et al. BioMagResBank. Nucleic Acids Research. 2008; 36
4. Hughes, John. Why Functional Programming Matters. The Computer Journal. 32 no. 2:1989.
5. McCarthy, John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM. Apr.1960
6. Backus, JW. The Syntax and semantics of the proposed international algebraic language of the Zurich ACMGAMM conference. http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf
7. Hutton, Graham; Meijer, Erik. Monadic parser combinators; Technical Report nottcs-tr-96-4, Department of Computer Science, University of Nottingham. p. 1996http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps
8. Spadaccini, Nick; Hall, Sydney R. Extensions to the STAR File Syntax. J Chem Inf Model. 2012; 52:1901–1906. [PubMed: 22725659]
9. sans: a library for parsing nmrStar files. https://octopus.bmrb.wisc.edu/svn/sans/
10. starlibj: a library for parsing NMR Star files. https://octopus.bmrb.wisc.edu/svn/starlibj/
11. Wattos software package. http://nmr.cmbi.ru.nl/~jd/wattos/
12. Torben Ægidius Mogensen. Basics of Compiler Design. http://www.itswtech.org/lec/dr.shaymaa/computation_theory_second/comptheory.pdf
13. Hall, Sydney R. The STAR File: A New Format for Electronic Data Transfer and Archiving. J Chem In5 Comput Sci. 1991; 31:326–333.
14. Wadler, Philip. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science. 1985; 201/1985:113–128.
15. Functional Java. http://functionaljava.org/
16. Guava: Google Core Libraries for Java 1.6+. http://code.google.com/p/guava-libraries/

```
   stop_

save_


##########################################
#  Molecular system (assembly) description  #
##########################################

save_system_melittin
   _Assembly.Sf_category                          assembly
   _Assembly.Sf_framecode                         system_melittin
   _Assembly.Entry_ID                             458
   _Assembly.ID                                   1
   _Assembly.Name                                 melittin
   _Assembly.BMRB_code                            .
   _Assembly.Number_of_components                 .
   _Assembly.Organic_ligands                      .
   _Assembly.Metal_ions                           .
   _Assembly.Non_standard_bonds                   .
   _Assembly.Ambiguous_conformational_states      .
   _Assembly.Ambiguous_chem_comp_sites            .
   _Assembly.Molecules_in_chemical_exchange       .
   _Assembly.Paramagnetic                         .
   _Assembly.Thiol_state                          .
   _Assembly.Molecular_mass                       .
   _Assembly.Enzyme_commission_number             .
   _Assembly.Details                              .
   _Assembly.DB_query_date                        .
   _Assembly.DB_query_revised_last_date           .

   loop_
      _Entity_assembly.ID
      _Entity_assembly.Entity_assembly_name
      _Entity_assembly.Entity_ID
      _Entity_assembly.Entity_label
      _Entity_assembly.Asym_ID
      _Entity_assembly.PDB_chain_ID
```

Figure 1.

An example section from an NMR-Star file showing its lexical and syntactic complexity. The keywords are ‘stop_’, ‘save_’, and ‘loop_’; comments begin with ‘#’; identifiers begin with an underscore; ‘save_***’ is a special token that identifies a save-frame; and the rest are called values. These are all examples of lexical units called tokens. Whitespace is not significant in most cases. The visible syntactic structures include key-value pairs, consisting of an identifier followed by a value; loops, consisting of a list of identifiers followed by a list of values, whose size is an integer multiple of the number of identifiers; and save frames, which consist of key-value pairs and loops.

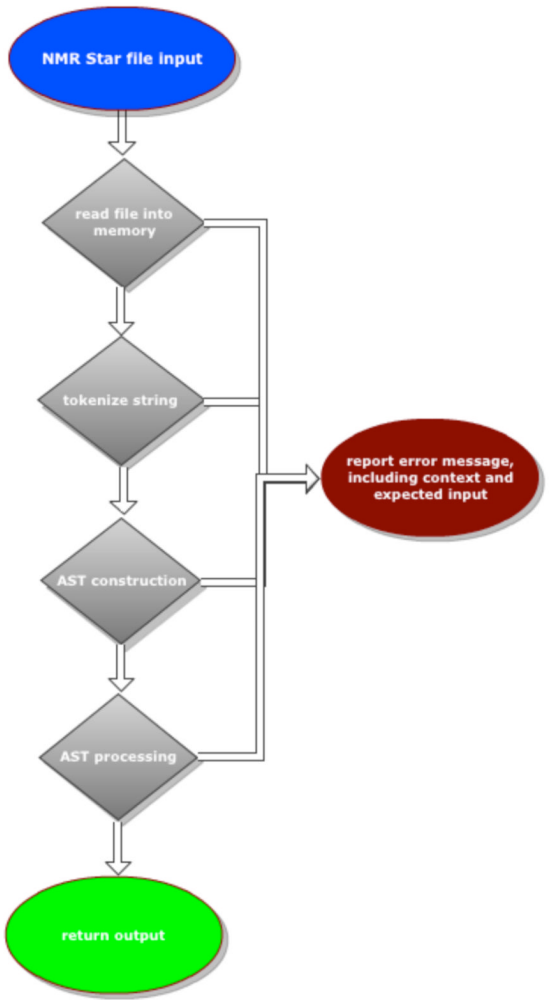

Figure 2.
The basic design of our parser. First, the input is read in. In step 2, it is lexically analyzed and broken into tokens. In step 3, the tokens are analyzed to construct an abstract syntax tree. In the last step, the AST is processed to convert the data to some other format or perform a data processing procedure. Note that any of the steps can fail, resulting in immediate aborting of the process. If processing fails, the parser attempts to provide the user with useful information as to where the error occurred and what happened.

```
Comment    = '#'  (not ('\n' | '\r' | '\f'))(*)

DataOpen   = "data_"  (not (' ' | '\t' | '\n' | '\r' | '\f' | '\v'))(+)

SaveOpen   = "save_"  (not (' ' | '\t' | '\n' | '\r' | '\f' | '\v'))(+)

SaveClose  = "save_"

Whitespace = (' ' | '\t' | '\v')(+)

Newline    = ('\n' | '\r' | '\f')(+)

Stop       = "stop_"

LoopOpen   = "loop_"

Identifier = '_' (not (' ' | '\t' | '\n' | '\r' | '\f' | '\v'))(+)

Value      =  single-quote-string | double-quote-string |
              semicolon-string | unquoted-string
```

Figure 3.

The NMR Star tokens in BNF-like format. Literal strings are surrounded in double-quotes, literal characters in single quotes, and standard character escapes are indicated by preceding a character with a '\'. Parenthesis indicates grouping, * means 'zero or more of', + means '1 or more of', '|' means either the left-hand pattern or the right-hand pattern must match, and adjacency means that first the left-hand pattern, then the right-hand pattern must be matched in sequence. The definitions of the four different ways of creating values can be found on our website.

```
starfile = data

data     = DataOpen save(+)

save     = SaveOpen (keyval | loop)(*) SaveClose

keyval   = Identifier Value

loop     = LoopOpen Identifier(+) Value(+) Stop
```

Figure 4.
The NMR Star syntactic structures in BNF-like format. Note that these definitions are in terms of the token definitions; token names begin with a capital letter. Note that comments, whitespace, and newline tokens do not appear here; they are filtered out and removed after lexical analysis and before syntactic analysis.