Logic Programming Technology For Interoperability between Bioinformatics Software Tools

J.R. Iglesias, G. Gupta, E. Pontelli, D. Ranjan
Dept. Computer Science
New Mexico State University
Box 30001/CS, Las Cruces, NM 88003
{jiglesia,gupta,epontell,dranjan}@cs.nmsu.edu

B. Milligan
Dept. of Biology
New Mexico State University
Las Cruces, NM 88003
brook@biology.nmsu.edu

Abstract

The goal of this project is to develop general solutions to enhance interoperability between different bioinformatics applications. Most existing applications adopt radically different formats and notations, forcing biologists into tedious translation work. The work reported in this paper is concerned with the use of Nexus as an intermediate data representation language. We develop (for the first time) a complete grammar for Nexus and we adopt this grammar to build a complete parser for Nexus. The construction heavily relies on the peculiar features of Prolog, to easily derive effective parsing and translating procedures. We also develop a general parse tree format which is suitable to support inter-conversion between Nexus and other formats.

Keywords: Nexus, Phylogenetic Inference, Domain-specific Languages, Logic Programming

Logic Programming Technology For Interoperability between Bioinformatics Software Tools

J.R. Iglesias, G. Gupta, E. Pontelli, D. Ranjan
Dept. Computer Science
New Mexico State University
{jiglesia,gupta,epontell,dranjan}@cs.nmsu.edu

B. Milligan
Dept. of Biology
New Mexico State University
brook@biology.nmsu.edu

January 1, 2001

Abstract

The goal of this project is to develop general solutions to enhance interoperability between different bioinformatics applications. Most existing applications adopt radically different formats and notations, forcing biologists into tedious translation work. The work reported in this paper is concerned with the use of Nexus as an intermediate data representation language. We develop (for the first time) a complete grammar for Nexus and we adopt this grammar to build a complete parser for Nexus. The construction heavily relies on the peculiar features of Prolog, to easily derive effective parsing and translating procedures. We also develop a general parse tree format which is suitable to support inter-conversion between Nexus and other formats.

1 Introduction

Information technology, powered by inexpensive and powerful computing resources, is increasingly becoming a critical component of research in many scientific fields. Despite this, information and computation technology has yet to realize its full potential, primarily due to the inability of scientists to easily accomplish the diversity of computational tasks required to solve high level scientific problems. For example, a typical bioinformatics analysis requires one to perform the following sequence of steps (as illustrated in Figure 1):

- Retrieve the desired sequences from a public molecular sequence database such as GenBank or GSDB, e.g., as a result of a BLAST query.
- Align the sequences using a program such as CLUSTAL W.
- Search for the best phylogenetic tree depicting the relationships among sequences using such programs as PHYLIP or PAUP.

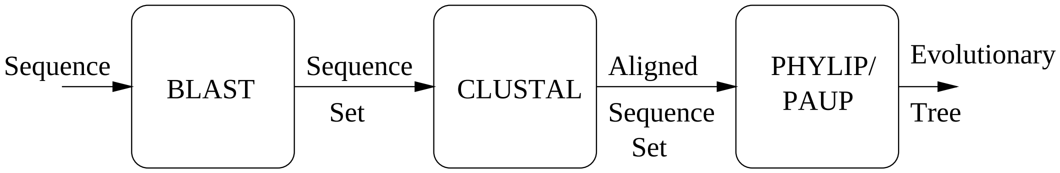

Figure 1: A Standard Phylogenetic Inference Task Pipeline

Although this sequence of tasks could be viewed as a single pipeline (see figure) through which data flows, in practice one immediately encounters a major difficulty with that metaphor. Specifically, these programs generally take input in a form not provided by the previous one and produce output incompatible with the next. Thus, between each of the scientifically relevant steps outlined above, translation from one file format to another is required in order to accomplish the main goal. Generally, unless the user is particularly adept at constructing scripts using shell tools (e.g., sed and awk) or programming in languages such as Perl and Python, this translation must be done manually, a tedious and error-prone procedure. Thus, computational biologists are faced with a large barrier when attempting to use available software tools: they must either devote significant energy in becoming computer programmers or they must spend endless hours editing, checking, and rechecking their data files when they convert from the output of one program to the input of another. Clearly, this is a very ineffective use of scientific talent.

Attempts have been made to develop general languages for the description of biological data (e.g., the recent efforts of the BioPerl group [2]). Of these proposals, the Nexus data description language [12] represents the most comprehensive effort to provide a universal, computer oriented language for the description of data for systematic biology applications. Nexus ideally encompasses all information a systematist or phylogenetic biologist may need (e.g., character data and trees). This makes Nexus the ideal data format for developers of new bioinformatics applications. In addition, the generality of Nexus makes it the the most viable alternative for a common intermediate data representation language, to be used as a communication bridge between bioinformatics applications, thus converting the ideal pipeline of Figure 1 into the actual pipeline depicted in Figure 2.

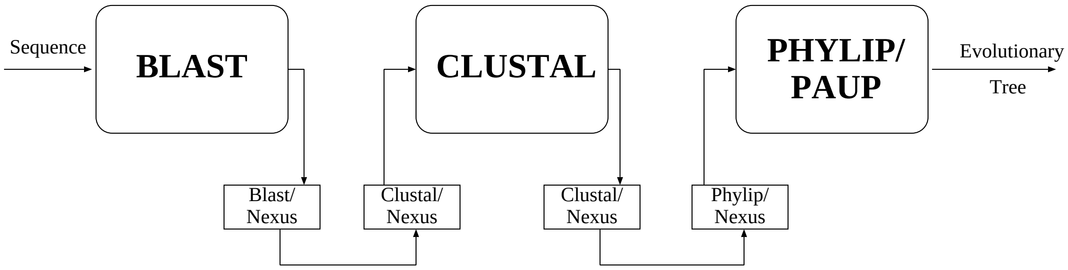

Figure 2: Modified Pipeline—using Nexus as Intermediate Language

In the last few years Nexus has gained considerable success, being adopted as input/output format in a number of phylogenetic inference applications. In spite of this, the number of available tools to recognize, parse, and manipulate Nexus files is really limited. Existing applications which make use of Nexus (such as PAUP and MacClade) do not provide independent tools for parsing Nexus. Furthermore, since these applications focus only on certain blocks of a Nexus files, their parsing capabilities are limited to such blocks. Most of these limitations, as it turned out from our experience, derive from the complex syntactic and semantic structure of the Nexus language—including very involved token definitions and a number of context-sensitive features—which makes the task of parsing an arbitrary Nexus file challenging.

The goal of this project is to develop general translation technology so that many of the important file formats can be inter-converted in a seamless way. We adopt Nexus as a common data representation language. To accomplish our goals we provide a formalization of the grammar for Nexus, and use it to build a parser for the complete Nexus language. To our knowledge, this is the first time that such an important task has been accomplished. This technology is based on logic programming techniques—the parsing features offered by Prolog, including *Definite Clause Grammars (DCGs)* and non-determinism, provide an excellent framework for tackling the complexity of Nexus (e.g., to handle the context-sensitive components of the language). Furthermore, Prolog's unification and the use of unification to provide general predicates with *reversible* behavior (i.e., arguments can be used to provide either input to or receive output from computations) makes our system easier to be used to provide inter-conversion services between different formats.

This project represents the first step towards a more ambitious goal: the development of a *domain-specific language* [3, 7] for describing biological processes and analyses of biological data at a high level of abstraction, and for creating reliable bioinformatics applications for a well-defined class of biological problems.

In this paper we present the results accomplished in the development of a formal specification of Nexus and its use to construct a complete parser for Nexus. We also exhibit a preliminary approach in using our system to provide inter-conversion services between Nexus and the PHYLIP [4] format.

2 Data Formats in Bioinformatics Applications

The abundance of multiple input/output formats in bioinformatics is witnessed by the fact that all the commonly used software packages produce as output and try to recognize as input a plethora of formats. For example, the output of a search of the genomic databases using BLAST (Basic Local Alignment Search Tool) [1] could result in many different forms of the output depending upon the the actual BLAST program (BLASTP, BLASTN, etc.) used and the choice of parameters for the BLAST search. Similarly, the histogram of expectations will be present or absent in the output of a BLAST search based on whether the `histogram` option was chosen or not. Similarly, CLUSTAL W [9] tries to automatically recognize 7 input formats for sequences—`NBRF/PIR`, `EMBL/SWISSPROT` , `Pearson (Fasta)`, `CLUSTAL (*.aln)`, `GCG/MSF (Pileup)`, `GCG9/RSF`, and `GDE` flat file. The output of CLUSTAL W can be in one of the five formats—`CLUSTAL`, `GCG`, `PHYLIP`, `NBRF/PIR`, and `GDE`. Although, the commonly used software packages try to recognize inputs in several formats automatically and even try to provide outputs in format suitable for commonly used bioinformatics tools, most of the time it is not possible to directly pass the output of one of these programs as input to another program. For example the typical output of a BLAST search can not be passed directly as an input to CLUSTAL W and the typical output of a CLUSTAL W run cannot be passed without modification as input to PHYLIP programs, unless the PHYLIP option was chosen in the CLUSTAL W run. CLUSTAL W does not provide an output option for PAUP. All this is unnecessarily inconvenient, restrictive and wasteful of computational resources as it implies that CLUSTAL W must be used more than once if different Phylogenetic Inference tools (say PHYLIP *and* PAUP) were to be used on the sequence alignment produced by it although the same alignment will be produced each time. One could avoid reusing CLUSTAL W if one had a fast translator from the PHYLIP input format to PAUP input format (which is the Nexus format [12]). Similarly, files in PAUP format produced during previous executions could be translated to PHYLIP format, to allow comparative inferences using different tools. This is illustrated in Figure 3.

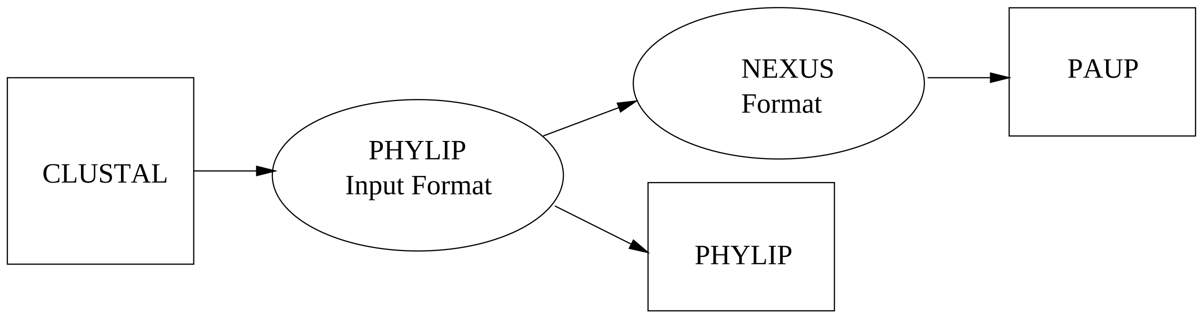

Figure 3: Avoiding Wasteful Computation via Translation

In addition to this, the fast development of the World-Wide Web has prompted the development of *markup languages* adequate to describe various classes of biological data—e.g., BioML (BIOpolymer Markup Language) [17] and BlastXML [22]. This introduces the additional need of inter-converting between these markup languages (currently used for data exchange purposes) and the more traditional formats recognized by existing applications.

One of the goals of this project is to provide the technology to build such fast translators quickly and reliably and use it to build the translators themselves.

The input/output format design for most bioinformatics software tools seems *ad hoc* with at best an informal description being available for what is acceptable as valid input and what output may be produced. Often, these are illustrated only via examples or explained informally and incompletely in English. The Nexus data description language perhaps is the most well-documented format of all, but, even for Nexus, a complete formal description is not available (for instance, the current version of the document at `Ag.Arizona.Edu/nexus/nexus.html` is a good example of the paucity of Nexus' description). A formal description of the valid inputs (and corresponding outputs) for the commonly used bioinformatics software tools, possibly via formal grammars, will go a long way towards format standardization and considerably ease translation between different formats. Providing such formal descriptions for inputs acceptable by various commonly used bioinformatics software tools is another important goal of this project. An important consequence of standardization is that biologists will be able to easily construct pipelines such as the one illustrated in Figure 1, and will be able to concentrate much more effort on their meaning of their analysis as opposed to the mechanics of performing it.

3 Approach to Formats Interoperability

3.1 Translators' Construction

Formal language translation refers to the task of translating a sentence written in a formal language $\mathcal{L}_1$ to an equivalent sentence in another formal language $\mathcal{L}_2$. Compilation is an instance of formal language translation where a program written in a high level language, say Java, is translated to an equivalent machine language program corresponding to the processor present in the computer. Thus, in this case the languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are Java and machine language, respectively, and the sentences are the programs.

One can build language translators in an *ad hoc* way for specific tasks. An *ad hoc* approach will work only for simple languages, or when a very small portion of a program/data is to be extracted and translated in some simple way to another language. However, when relatively complex languages are involved, this approach is hard, time consuming, and unreliable. For many applications, one needs to translate a complex language into another. An example of this is translating the output of alignment program CLUSTAL W to the Nexus format that is suitable for use by PAUP. It is possible to write an *ad hoc* translator for each specific program. This is indeed what has been done in most existing applications which make use of the Nexus file format. However, this approach is not scalable in the sense that for each software package an *ad hoc* translator has to be developed. This results in a lot of repeated effort as the commonality among these different translators is not fully exploited.

It is commonly agreed that breaking up the translation process into *lexical analysis*, *parsing*, and *generation* phases is the best approach to take if one wants to build reliable, robust, flexible, and extensible translators. Lexical analysis refers to analyzing the tokens from the raw character stream constituting the program. Parsing refers to inferring the structure among the stream of tokens, while generation refers to the actual task of converting the inferred structure to the target language format.

Lexical analysis is a relatively easy task; however, the complexity of the parsing and generation procedures depends on the complexity of the languages involved. Parsing in itself is quite a hard task. Two approaches have traditionally been used to build parsers.

1. Develop a parser from scratch for a given language; i.e., write programs to recognize the structure among the tokens, which may involve implementing all sorts of state machines and automata. This is a time consuming task and may produce unreliable parsers. Resulting parsers are always hard to verify.

2. Use available tools to automatically generate parsers. One commonly available tool is YACC/Bison. However, YACC/Bison can handle only certain restricted types of languages (those for which a *context free LALR(1)* grammar can be written). Thus, developing a complete translation system using YACC/Bison is still a non-trivial task, since a context-free LALR(1) grammar has to be designed first for the language. There are various problems that one might encounter, including the fact that an LALR(1) grammar for the given language may be hard to construct, or occasionally even impossible.

Of the above two approaches, the second one is considerably less complex, more reliable and theoretically better grounded. Hence it has been more popular and successful. However, the issues discussed above indicate that building a parser using YACC/Bison for a non-trivial language is still a difficult task that lies outside the domain of expertise of most biologists. Even if one has the expertise, developing a YACC/Bison based parser and generator is a time-consuming task and requires consideration of many tedious details.

Furthermore, while neither of these approaches can be generalized to accommodate additional translators or language extensions (they are, after all, fundamentally "one-off" activities), both require the same careful analysis of the structure of the language being translated as the more general approach we advocate below. They also require much greater investment in verification. Thus, it is illusory to expect that one can save effort, especially in the long run, by adopting either of the above approaches to developing translation technology that will enable pipelines such as the one depicted in Figure 3.

3.2 Translation Approach

To achieve our goal, the methodology that we use should be general enough so that the task of format conversion is facilitated. This methodology should be extensible, in the sense that if a new format is introduced, then the amount of work required to develop conversion software w.r.t. the other formats is minimal.

To accomplish both these objectives we propose to make use of a common intermediate data description language, general enough to subsume all popularly used existing formats for biological software systems (e.g., PHYLIP, PAML, CLUSTAL). The advantage of following this approach is that we just need to build two converters for every format, F, one for converting F to the common format and another one to convert the common format to F.

In the first stage of this work, the choice of the common intermediate language fell on Nexus [12]. Nexus represents a good starting point since

- it has been originally designed as a data description language, independent of any specific operating environment or specific application, and general enough to encompass the most relevant data types used in bioinformatics applications;
- it is provided with a semi-formal language description, intentionally oriented towards computer processing.

Nexus, in its current format, however, is still limited in that it can represent biological information, but cannot represent format-related information, such as those adopted in the outputs of some popularly used biological software systems (e.g., CLUSTAL W). On the other hand, the design of Nexus envisions the possibility of extending the language with additional features (e.g., application dependent components), which can be developed to cover these missing capabilities.

3.3 Logic Programming Technology

In this project we propose to use *logic programming* [19], and more specifically its implementation in the Prolog language, for the development of translation tools between Nexus and other data formats. This leads to solutions that are less time-consuming, more reliable, flexible and extensible. The *Definite Clause Grammar (DCG)* facility of Prolog allows one to rapidly specify and implement parsers. Logic programming can also be used for rapidly developing back-end generators. Thus, with the use of logic programming, a translator can be quickly developed.

We believe that one of the reasons for a paucity of translators and parsers for biological notations and languages is because of the complexity of the traditional translation technology (e.g., we could not find a complete public domain parser for Nexus on the web). The reason translators can be rapidly developed using the logic programming approach is because a logic programming language (e.g., Prolog) is a very high level, declarative language in which users can write programs at a very high level of abstraction.

In addition to the general high-level and declarative nature of the language, there are a number of specific features of Prolog that have lead us to select it as development platform for this project:

1. as mentioned earlier, Prolog offers Definite Clause Grammars, where a standard grammar-based specification of a language can be turned into an actual parser with minimal effort. Since our work on parsing Nexus needs to start from the formalization of the language itself, the use of DCGs allows us to combine the two phases to incrementally construct a working parser as we go along with the formalization of the language.
2. Prolog's non-determinism (the ability to automatically explore alternative computations in search of a solution) allows one to produce (from a DCG specification and with no extra effort) a non-deterministic parser. Since the languages we are dealing with contain, from the parsing point of view, very complex features, the use of non-determinism allows us to obtain a working parser without the need of modifications to the grammar;
3. the ability to add *arguments* to the components of the grammar allows us to create "communication channels" between the grammar elements (similarly to what happens with attribute grammars), allowing to handle with ease some of the context-sensitive components encountered in Nexus.

4. the use of unification as an argument passing mechanism, allows one to build "reversible" procedures, i.e., procedures whose arguments can be used either as input or as output depending on the needs. In the context of programs for formats translation, it is possible to develop a single program which translates between two given formats and use it to perform translation in both directions (from the first format to the second and vice versa).

In order to proceed in a systematic fashion in the development of our framework, we have taken inspiration from the work on *Horn Logic Denotational Semantics* [6]. In this framework the denotational semantics of a formal language (domain specific or a traditional one) $\mathcal{L}$ is given in terms of Horn clause logic (or logic programming). Translators can be easily obtained by providing the semantics of one format L_1 in terms of another format L_2—i.e., L_2 becomes the semantic algebra used to provide the semantics of L_1. The encoding of these semantics specifications in terms of logic programs allows one to automatically obtain a provably correct and working translator.

4 Nexus

Nexus [12] is a data description language created as an attempt to provide biologists practicing phylogenetic and systematic biology with a common and application independent data format. The main principles behind the design of Nexus are:

- *processibility:* differently from many of the other formats, the design of Nexus promotes (semi-)formal specification to facilitate the automatic processing of Nexus files by computer programs;
- *expandability:* the structure of Nexus files is modular, allowing the addition of new types of data blocks without excessive disruptions to the rest of the file structure;
- *inclusivity:* the choice of data types provided in Nexus is expected to cover all the needs of typical phylogenetic bioinformatics applications;
- *portability:* while most of the existing data formats have been designed to fit the needs of a specific application, Nexus is an application independent language; its specification attempts also to accommodate for operability within different computing environments.

Each Nexus file is composed of a number of *blocks*, and each block is *typed*—i.e., it is used to describe a specific type of data (e.g., *Taxa*, *Characters*, *Codons*). In addition to these common blocks (called *public blocks* in Nexus terminology), Nexus allows also for *private blocks*, which are meant to encapsulate additional program dependent data—e.g., the *MacClade* block is used to represent additional data components required by the MacClade system [13].

Figure 4 shows a simple Nexus file with two blocks (a more complex example is presented in the next section). Each block is composed of *commands*; each command is used to specify a certain property of the collection of data described by the block. For example, in the code in Figure 4, the block `CHARACTERS` describes the characters observed for a given set of taxa; the command `CharStateLabels` identifies the labels describing the characters of interest and their possible values, while the `Matrix` command associates to each taxon the corresponding character values.

Nexus has gained considerable popularity in recent years. An increasing number of phylogenetic applications (e.g., PAUP 3, MacClade, COMPONENT) have adopted Nexus as their input/output data format. Furthermore, Nexus has also been adopted as a format for data housing in a number of public databases (e.g., TreeBase, Genetic Data Analysis).

5 System implementation

5.1 Formal Description of Nexus

A Nexus file, taken from [8] with some modifications, is shown in Figure 6. Any file must start with the token `#nexus` and the information is organized in blocks delimited by the words `begin` and `end`. The name of a Nexus block,

```
#Nexus
begin CHARACTERS
   DIMENSIONS NChar=3;
   CharStateLabels 1 hair/absent present,
                   2 color/red blue,
                   3 size/small big;
   Format Tokens;
   Matrix
        taxon_1 absent red big
        taxon_2 absent blue small
        taxon_3 present blue small;
end;
begin Trees;
     Tree best = (fish, (frog, (snake, mouse) ) );
end;
```

Figure 4: Sample Nexus File

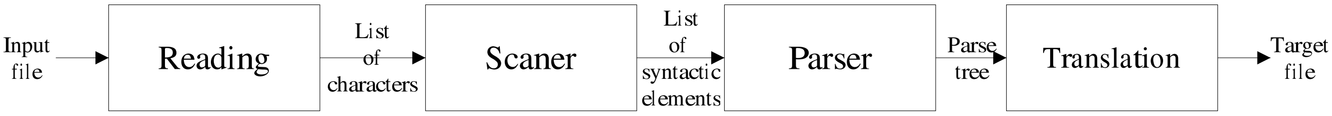

Figure 5: Translation Process

comes right after the `begin` word. For instance, in Figure 6 there are four Nexus blocks, namely `data`, `codons`, `assumptions`, and `paup`. Nesting blocks inside another Nexus block is not allowed. Each block is composed of several commands and each command may have several options. In Figure 6 `exset` and `charset` are commands in the `assumptions` block. Similarly, the `format` command in the `data` block has two options, `datatype` and `interleave`.

A formal description of Nexus has been attempted in [12]. In spite of the effort, the outcome is insufficient to be directly usable in the generation of a parser for Nexus. The description provided in [12] is not based on formal language constructions (e.g., grammars) but in terms of templates and examples. As a result:

- the specification is largely incomplete and contains a number of undefined or partially inconsistent situations;
- the lack of a formal language specification at the time of the design of Nexus has led to a language containing a number of features which are difficult to specify and even harder to parse.

Even though, the general structure of a Nexus file is well defined, its syntax has several characteristics that makes it different from the syntactic constructs commonly found in programming languages. Moreover the current Nexus standard is quite large. To accomplish the goals of this project we have developed a complete BNF grammar for the Nexus language. This has been accomplished by combining the semi-formal specification presented in [12] with information drawn from user manuals of various applications which use Nexus (e.g., PAUP) and various data files obtained from the Web and from local biologists. Our BNF grammar definition of the Nexus standard [12] is around four thousand lines long, and this will grow bigger as Nexus standard is extended by adding more blocks or commands. Some of those possible extensions are already implemented in some new versions of MacClade [13].

In order to translate from a Nexus file to another file format, our program follows the traditional four phases (see Figure 5). First, the system reads the input file in a list of character codes. Second, the list is scanned creating a list of tokens. After this, the list of tokens is parsed using a *Definite Clause Grammar* (DCG) to create a parsing tree which in turn is suitable for translating from Nexus to other formats. We shall devote the remaining of this section to describe each of these phases emphasizing the techniques applied during the implementation.

```
#NEXUS
[!Primate mtDNA]
begin data;
    dimensions ntax=12 nchar=898;
format datatype=dna interleave gap=-;
    matrix
Homo_sapiens        AAGCTTCACCGGCGCAGTCATTCTCATAATCGCCCACGGGCTTACATCCT
Pan                 AAGCTTCACCGGCGCAATTATCCTCATAATCGCCCACGGACTTACATCCT
Gorilla             AAGCTTCACCGGCGCAGTTGTTCTTATAATTGCCCACGGACTTACATCAT
Pongo               AAGCTTCACCGGCGCAACCACCCTCATGATTGCCCATGGACTCACATCCT
Hylobates           AAGCTTTACAGGTGCAACCGTCCTCATAATCGCCCACGGACTAACCTCTT
Macaca_fuscata      AAGCTTTTCCGGCGCAACCATCCTTATGATCGCTCACGGACTCACCTCTT
M._mulatta          AAGCTTTTCTGGCGCAACCATCCTCATGATTGCTCACGGACTCACCTCTT
M._fascicularis     AAGCTTCTCCGGCGCAACCACCCTTATAATCGCCCACGGGCTCACCTCTT
M._sylvanus         AAGCTTCTCCGGTGCAACTATCCTTATAGTTGCCCATGGACTCACCTCTT
Saimiri_sciureus    AAGCTTCACCGGCGCAATGATCCTAATAATCGCTCACGGGTTTACTTCGT
Tarsius_syrichta    AAGTTTCATTGGAGCCACCACTCTTATAATTGCCCATGGCCTCACCTCCT
Lemur_catta         AAGCTTCATAGGAGCAACCATTCTAATAATCGCACATGGCCTTACATCAT
    ;
end;

begin codons;
    codonposset * codons =
        N: 1 458-659 897 898,
        1: 2-455\3 660-894\3,
        2: 3-456\3 661-895\3,
        3: 4-457\3 662-896\3;
    codeset * codeset = mtDNA.mam.ext: all;
end;

begin assumptions;
    usertype ttbias (stepmatrix) = 4
           A C G T
       [A] . 6 1 6
       [C] 6 . 6 1
       [G] 1 6 . 6
       [T] 6 1 6 . ;
    charset tRNA_His = 459-528;
    charset 'tRNA_Ser_(AGY)' = 529-588;
    charset 'tRNA_Leu_(CUN)' = 589-659;
    charset 1st_positions = 2-455\3 660-894\3;
    charset 2nd_positions = 3-456\3 661-895\3;
    charset 3rd_positions = 4-457\3 662-896\3;
    exset  protein_only = 1 458-659 897 898;
    exset  non_protein = 2-457 660-896;
end;

begin paup;
    [Standard ML benchmark]
    outgroup Lemur_catta Tarsius_syrichta;
    set criterion=likelihood;
    lset var=f84;
    hs;
end;
```

Figure 6: A Sample Nexus File

```
[
    ['#NEXUS',[word,unquoted],1,1],[['\n'],[new_line],1,8],
    ['[!PrimatemtDNA]',[comment,comment_command],2,1],['\n',[new_line],2,17],
    ['begin',[word,unquoted],3,1],['data',[word,unquoted],3,7]['; ',[symbol],3,11],

   ...

]
```

Figure 7: Result of Lexical Analysis

5.2 Scanning

The scanner of a Nexus file is quite unusual in many ways because of the syntactic and semantic properties of the tokens. The Nexus format is non-case sensitive. Comments in Nexus are enclosed in square brackets `[ ]` and comments can be nested. For instance, `[[[valid]]]` is a valid comment, while `[[[invalid]]` is not. Contrary to most usual languages, comments should not be discarded during scanning for two reasons. First, a comment can be a *comment-command*, that is, a comment that contains a command which may be used by an specific program. Second, comments in a Nexus file usually contain important scientific notes which are specific to the data, and without them the data may loose some relevant information. It is worthwhile to mention that the format of a comment-command is very general and particular to each program, so what for some program can be a comment that can be discarded, for some other program might mean essential information to complete a genetic analysis.

A token in Nexus is basically a sequence of characters delimited by space or punctuation symbols. However, there are some important exceptions. A token can include spaces and punctuation if it is enclosed in quotation marks, and *comments do not break tokens*. For example,

`drosophila[\b]' aspartic'[\i]' acid'[\p]` (1)

represents the single token `'drosophila aspartic acid'`, and the comments in (1) are in fact comment-commands indicating that `aspartic` should be displayed in bold, `acid` should be displayed in italics, and the following tokens should be displayed using plain letters. One more *feature* of Nexus is that an underscore in a non-quoted token must be interpreted as a blank space. Therefore, `DROSOPHILA_ASPARTIC_ACID` and the token in (1) represent exactly the same syntactic object.

Another interesting property in the tokens on a Nexus file is that the scanner must be able to recognize numbers in the standard integer or real format, but an identifier can start with a number. For example, `0.339e-4_to_1.23` should be recognized as a valid identifier. Finally, there is no strict notion about *reserved-words* in the Nexus standard and most tokens have different meaning depending on the file context. For example, `begin` is a token that may be used to delimit the beginning of a new Nexus block, but inside a block `begin` can be used as a valid identifier. Similarly, `2.34` in a data matrix can represent either the real number of a taxon frequency, or the symbols `2 . 3 4` corresponding to a DNA sequence. Besides, some single characters, like the newline character, may or may not have a significant syntactic meaning depending on the context or according to several selected options by commands in the file.

Due to the characteristics of Nexus, just a few token recognition decisions can be made during the scanning process and even some of them have to be delayed to the translation in the parsing phase. Our approach to solve this problem is to recognize tokens in a general manner assigning in many cases more than one possible type to a recognized token. The result of the scanning process is a list of *syntactic elements*, like the one in Figure 7. The figure shows a formatted fragment of the list of syntactic elements produced by the scanner taking the file in Figure 6 as input. The token items, i.e., the first element of each syntactic element, is a list of character codes in real life but here we present them as a single character string for the sake of readability. Each syntactic element is in turn a list with four items `[Token, Type, Line, Character]`. The item `Token` is a list of character codes representing the token. The second item, `Type`, is a list of possible types for that token. The third and fourth items represent the line and character coordinates of the first character of the token in the file. The token coordinates are used during the parsing and translation phase to communicate possible warnings, errors, or some other messages to the user.

Note that in Figure 7 newline characters and comments are not discarded for the reasons exposed above. Also some tokens may have more than one recognized type. For instance the fourth token has two recognized types, `comment` and `comment_command`. During parsing and translation, the correct type expected is matched with any of the possible types recognized in the scanning phase. These activities have been highly facilitated by the use of Prolog, with its non-deterministic search capabilities and the flexible use of logical variables. As an interesting example, consider again the token in (1). Recall that comments should not be discarded during the scanning phase. The corresponding syntactic element created for this token looks like:

```
[[ ['Drosophila'],[word,unquoted],1,1],
   ['[\b]]',[comment,comment_command],1,11],
   [''' aspartic''',[word,quoted],1,15],
   ['[\i]',[comment,comment_command],1,26],
   [''' acid''',[word,quoted],1,30],
   ['[\p]',[comment,comment_command],1,37]
 ],
 [word,comment_word],1,1]
```

In this case the type `[word, comment_word]` is used to clarify that this token is a word composed of comments and words. The corresponding token item of this syntactic element contains a list of syntactic elements which may have to be put together in the parsing or the translation phase when this is required.

Related also to the issue of handling comments, Nexus allows a number of special type of comments to affect the parsing process as well. For example, the use of the comment-commands `[&U]` and `[&R]` in a `Tree` block have to be recognized by the parser, since they are used to communicate whether the described tree is *unrooted* or *rooted*.

5.3 Parsing

Once a list of syntactic elements is created in the scanning phase, parsing is straightforward using Prolog's built in DCGs. However, there are several aspects that had to be taken into account to create the Nexus parser. First, Nexus is a format that is widely used by several programs, but it is also under continuing development. Some syntactic elements in original Nexus blocks have been dropped and some others have been added. Since many files were written using seminal Nexus formats, the current standard [12] encourages support of every version of Nexus if possible. In addition, further enhancements are planned in future Nexus releases.

In order to meet these requirements, Prolog has been very useful in terms of program organization and maintenance. The Nexus parser is in fact composed of several parser modules, one for each different Nexus block. Each parser is written with an independent DCG. In this manner recognizing a new Nexus block in the future would consist of just adding a new parser module with a DCG capable of parsing the new block. Moreover, future standard modifications to a given block should imply only modifications to the corresponding parsing module.

A more interesting aspect in Nexus is that it allows the introduction of any new user-defined Nexus block, or even the introduction of any new user-defined command inside a standard Nexus block. In this regard, the standard [12] suggests that during parsing, a non-recognized structure should be skipped sending a corresponding warning message. However, a Nexus file should be rejected with an error message when a grammatical structure *seems* to follow a standard pattern but at some point a syntactic problem is found producing an ambiguity that makes it impossible to recognize the entire file.

Further complications arise from the presence of context sensitive features in the structure of the data descriptions. A typical example is represented by the interaction between values of certain tokens acquired from the input and the structure of the successive input string. For example, the Nexus language allows one to write:

```
Begin Data;
 DIMENSION NCHAR = 7;
 FORMAT DATATYPE = DNA MATCHCHAR .;
 MATRIX
   taxon_1  GACCTTA
   taxon_2  ...C..A
   taxon_3  ..C.T..
End;
```

The matching character "." indicates that the same character in the same position as the previous taxon is to be used. This feature of Nexus cannot be coded as a YACC grammar rule. Additionally, the fact that one is allowed to declare the number of characters in the sequences `taxon_1`, `taxon_2`, and `taxon_3` by using the command `DIMENSION NCHAR = 7` makes the language context sensitive.

With respect to these last requirements, Prolog has been a valuable implementation tool. The embedded non-deterministic behavior of a DCG allows us to *guess* the grammatical structure of the block or the command. Whenever a standard block or a command pattern is found, the parser either recognizes it or rejects it depending on whether a syntactic problem occurs in the file. On the other hand, if the parser is unable to find a standard pattern in the current grammatical structure, such structure is accepted as unrecognized and a warning message is sent informing the user about this action.

There is a final implementation detail in the parser that is important to describe. A DCG assumes that the the input to parse is represented as a difference list of tokens. Tokens in the difference list are matched against terminal symbols in a DCG rule using unification. However, the input we use to a DCG is a list of syntactic elements (see section 5.2), and unification does not work any longer as a token matching mechanism, due to the matching requirements described in section 5.2.

Recall that a terminal symbol in a DCG is denoted as a list, that is, enclosing the terminal symbol in square brackets. For example, using a standard DCG, the main Nexus DCG rule in the parser would look like:

```
nexus --> ['#nexus'], blocks.                (2)
```

In our approach, the normal terminal symbol definition is replaced in the obvious way by a call to

```
match(?Token,?Type,-Line,-Character,-Comments,+Begin,-End)
```

where `Begin` and `End` represent a difference list of syntactic elements. For example, the DCG rule in 2 is in written in the parser as:

```
nexus( nexus(Blocks,Comments) ) -->
  match('#nexus', Type, Line, Character, Comments)
  blocks(Blocks).
```

In this case `match` succeeds if `Token` matches with the token in the head of `Begin` while `Type` contains the type of `Token` recognized by the scanner, `Comments` is the list of all comments preceding `Token`, and the coordinates of the token are stored in `Line`, `Character`. All this information is usually stored in the generated parse tree and used in the translation phase to facilitate the conversion process, report translation errors, or handle the comments in a convenient way according to the target format specification.

5.4 Translation

A parse tree suitable for translation is the result of the parsing process. In the translation phase, the parse tree is interpreted and the result of that interpretation is written in the corresponding target file format. The system is intended to be able to translate from Nexus to other several file formats so, as in the parser, the approach we follow is modular in the sense that a different Prolog module is provided for each target format supported. Therefore, adding support for a new target format should imply just adding a new corresponding Prolog module. As stated before, we expect that this modularization strategy shall simplify both future development and maintenance.

The translator module is implemented in a straightforward manner [7, 6]. The parse tree is interpreted in a top-down fashion using DCG rules with a format similar to the DCG rules used in the parser. As a simple example, the translation rule looks like the following:

```
ph_nexus( nexus(Blocks), Store ) -->
   read(Store, output_file, FileName),
   open(FileName, write, Handle),
   write(Store, output_handle, Handle),
   ph_blocks(Blocks, Store),
   close(Handle).
```

Note the one to one correspondence between this translation rule and the parsing rule described in section 5.3. The `ph_` prefix is used to denote that this rule is a PHYLIP translation rule. Similar prefixes are planned for each different translator module to be supported in the future.

The variable `Store` is used to keep the computation state during the interpretation. The predicates `read(+Store, +Location, -Value)` and `write(+Store, +Location, +Value)` are used respectively to get the value of some location in the store, or to write a value to a location in the store. The internal format of a store is very simple, a `Store` is just a list of pairs `[Location, Value]`. Additional side-effect predicates, like `open` and `close` in the previous example, are inserted in the translation rules as necessary to accomplish the final translation.

6 Discussion

All the components in our system are currently implemented in Prolog, which may represent both some advantages and challenges. It is well known that LR(k)-parsers have important advantages over other kinds of parsers, like fast parsing and reduced use of stack memory. Moreover, LALR(1) parse generators and optimized-scanner generators are already available for other programming languages, e.g., YACC and LEX for the C language.

On the other hand, the declarative nature of Prolog usually provides a faster implementation, a small programming code, and more confidence about the correctness of the program. Further, the inherent nondeterminism of a DCG is particularly useful for languages where ambiguity or exceptional events may arise during parsing, like in Nexus.

In our case the most time consuming part of creating a Nexus parser was to come up with a BNF description of the Nexus grammar. Unfortunately, the description given in the standard [12] is not formal at all – that may be one reason of why there is no other fully compliant Nexus standard parser to the best of our knowledge. Our original BNF description of Nexus was intentionally done using a syntactical form which is very close to a DCG. Therefore, once we constructed the BNF we essentially had the parser completed. Practically, the only modifications to the code with respect to the BNF description is related to the special match routine (see section 5.3) and the extra argument in the DCG rules required to generate the parse tree. The quantity of code used in the final parser is close to the size of the BNF description. More important is the fact that we were quite certain about the behavior of the parser, since the BNF description was carefully reviewed.

With respect to the scanner generator issue, it is necessary to mention that there also exist some publicly available scanner generators for Prolog, e.g., Elex(Prolog) and PLEX [18, 21]. These scanner generators may not be as fast as a scanner generated by LEX, but they at least relieve the programmer of the task of writing and maintaining the scanner by hand. In particular, Elex(Prolog) seems to be quite flexible and has a very acceptable performance, especially in cases where the finite automata can be defined in a deterministic way so that the internally implemented *look-ahead* feature in Elex(Prolog) is not necessary. In this regard we should remark that the logic programming community would benefit from the development of an optimized-scanner Prolog generator like LEX, especially knowing that compilers, interpreters, and natural language processing systems are fertile areas for Prolog.

Some other features in Prolog provide pleasant development and run-time environments. Different parsers and translators specific to some parts of the whole grammar can be kept in separate modules facilitating future development and maintenance. Additionally, parsing and translation modules can be added on-the-fly at run time by just loading the corresponding module.

As a final note in favor of using Prolog and DCGs, we should say that given the computing power of today's computers, it is very unlikely that the final program can run out of memory or run with an unacceptable time performance. We have been able to parse Nexus files as large as 600KB in approximatively 15 seconds[1].

7 Related Work

As mentioned earlier, very limited effort has been invested in the development of *general* tools for handling Nexus files. Most of the applications which are making use of the Nexus format limit their attention to recognizing only specific parts of Nexus files, thus not providing complete parsers for the language.

[1]This was measured in a 466Mhz Linux PC

The work that comes closest to ours is represented by *NCL* [11], which is a C++ Class Library for reading Nexus files. The library deals properly with the syntax of Nexus (including the various uses of comments), but does not have a complete coverage of the language (e.g., there are limitations in recognizing CHARACTERS blocks). The use of C++ libraries offers the potential of extending NCL to recognize new blocks (e.g., private blocks), by deriving new subclasses. On the other hand, the support for the creation of these new classes is limited (only in terms of access to the lexical analyzer). This clearly does not match the simplicity of extending our system, since the grammar is explicitly present (as a DCG) and the structure considerably more modular.

A Windows-based tool called *NDE* [15] has also been developed to create and edit Nexus files using a spreadsheet-type interface. The tool has the ability to attach user annotations (even in the form of pictures) to components of the file, but has limited parsing capabilities; being developed mostly as an editor to manually create new Nexus files, NDE places various limitations on the kind of Nexus files it is capable of reading. Furthermore, NDE does not provide any way of using its parsing capabilities for format interconversion.

The use of logic programming as a technology to enhance interoperability between software applications and data formats has recently gained relevance in a number of application domains. Several examples have been demonstrated in the context of Web applications [16] and in the conversion between different formats for mathematical entities (with particular focus on the the issue of math accessibility for visually impaired [10]).

8 Conclusion and Future Work

In this paper we have described our work in constructing a system for recognizing, parsing and translating the Nexus data representation language. Nexus is a universal language for the representation of data for phylogenetic and systematic biology applications. The task has not been easy. Nexus is a rather informally specified language, and, in spite of the effort of its designers, it contains a number of features that are hard to describe and even harder to parse.

Our task has been accomplished by first constructing a complete grammar for the Nexus language. The grammar has allowed us to easily derive a parser for Nexus, by directly translating the grammar rules into Prolog's DCG rules. The definition of the language turned out to contain a rather complex definition of tokens and a number of context-sensitive features. Prolog's ability to perform non-deterministic parsing, and the ability to use DCGs to perform context-sensitive parsing, allowed us to overcome these difficulties without much effort.

We have also used our system to produce the first of a series of inter-conversion programs, allowing translation between Nexus and other commonly used formats. Indeed, the overall objective of our work is to use Nexus as a intermediate language to provide interoperability between a number of different data representation formats. At this time we have successfully experimented with the inter-conversion between Nexus and the PHYLIP format.

The results described in this paper represent the first step of an ambitious project, called ΦLog. The goal of this project is to develop a declarative, domain-specific language, designed to provide biologists with a convenient tool for designing phylogenetic applications (without the need of any specialized programming background). In this first phase we have focused on the use of Nexus as a common data description language—in this perspective Nexus is an excellent domain-specific data description language. What is missing in Nexus is the ability to describe *computations*. Our objective is to extend Nexus by providing it with a number of primitive operations corresponding to the most basic operations required during phylogenetic inference processing, and a number of declarative combinators to combine primitive operations and describe complete inference processes. This is akin to allowing high-level descriptions of pipelines such as the one depicted in Figure 1. The beauty of this approach is that the mapping between the high-level steps of the inference process and the corresponding applications in charge of executing such steps (e.g., CLUSTAL W for sequence alignments, PAUP for tree inference, etc.), as well as all the interoperability issues (e.g., conversions between input/output formats) will be completely transparent to the domain-specific language programmer (e.g., a biologist).

Acknowledgments

The authors wish to thank J. Spalding and P. Lammers for the discussions related to this project. This work is partially supported by a grant from the National Biotechnology Information Facility Project, and NSF grants

CCR9875279, CCR9900320, CDA9729848, EIA9810732, CCR-9820852, and HRD9906130.

References

[1] S.F. Altschul and B.W. Erickson. Significance of nucleotide sequence alignments: a method for random sequence permutation that preserves dinucleotide and codon usage. *Mol. Biol. Evol.*, 2:526–538, 1985.

[2] BioPerl. *BioPerl Project: XML*. `bio.perl.org/Projects/XML`.

[3] C. Consel. Architecturing software using a methodology for language development. In *Proc. Int'l Symp. on Prog. Lang. Impl., Logics and Programs*, pages 170–194. Springer LNCS 1490, 1998.

[4] J. Felsenstein. PHYLIP: Phylogeny inference package, version 3.5c, 1993.

[5] GenBank Overview. `www.ncbi.nlm.nih.gov/Genbank`.

[6] G. Gupta. Horn Logic Denotations and Their Applications. In *The Logic Programming Paradigm: The next 25 years*, Springer Verlag, May 1999, pp. 127-160.

[7] G. Gupta and E. Pontelli. A Horn logic denotational framework for specification, implementation, and verification of domain specific languages. Technical report, NMSU, March 1999. `www.cs.nmsu.edu/~gupta/sem.html`.

[8] K. Hayasaka, T. Gojobori, and S. Horai. Molecular phylogeny and evolution of primate mitochondrial DNA. *Mol. Biol. Evol*, 5:626–644, 1988.

[9] D. G. Higgins, J. D. Thompson, and T. J. Gibson. Using CLUSTAL for multiple sequence alignments. *Methods in Enzymology*, 266:383–402, 1996.

[10] A. Karshmer, G. Gupta, S. Geiger, C. Weaver. Reading and Writing Mathematics: the MAVIS Project. *Behavior and Information Technology*, 18(1), 1999.

[11] P.O. Lewis. NEXUS Class Library (NCL) Release 2. `lewis.eeb.uconn.edu/lewishome/ncl2`.

[12] D.R. Maddison, D.L. Swofford, and W.P. Maddison. NEXUS: An Extensible File Format for Systematic Information. *Syst. Biol.*, 464(4):590–621, 1997.

[13] W.P. Maddison and D.R. Maddison. *MacClade: Analysis of phylogeny and character evolution, version 3.07.* Sinauer, Sunderland, Massachusetts, 1997.

[14] National Center for Genome Resources. Genome Sequence Database (GSDB). `www.ncgr.org/research/sequence`.

[15] R.D.M. Page. A NEXUS Data Editor for Windows. `taxonomy.zoology.gla.ac.uk/rod/NDE`.

[16] E. Pontelli, G. Gupta, et al. A Domain Specific Language Framework for Non-Visual Browsing of Complex HTML Structures. *Int. Conf. on Assistive Technologies*, ACM, 2000.

[17] Proteometrics, LLC. The BIOpolymer Markup Language (BIOML). `www.bioml.com/BIOML`.

[18] P.B. Reintjes. MULTI/PLEX: An AI tool for formal languages. Unpublished, 1994.

[19] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1996.

[20] D. L. Swofford. PAUP: phylogenetic analysis using parsimony, version 3.0. Technical report, Illinois Natural History Survey, 1989.

[21] G. van Noord. Prolog(Elex): A new tool to generate Prolog tokenizers. Unpublished, 1997.

[22] WorkingObjects.com BlastXML. `workingobjects.com/blastxml`.