Interoperability between Bioinformatics Tools: A Logic Programming Approach

J.R. Iglesias[1] and G. Gupta[3] and E. Pontelli[1] and D. Ranjan[1] and B. Milligan[2]

[1] Dept. Computer Science, New Mexico State University
[2] Dept. of Biology, New Mexico State University
[3] Dept. of Computer Science, UT Dallas
{jiglesia,gupta,epontell,dranjan}@cs.nmsu.edu, brook@biology.nmsu.edu

Abstract. The goal of this project is to develop solutions to enhance interoperability between bioinformatics applications. Most existing applications adopt different data formats, forcing biologists into tedious translation work. We propose to use of Nexus as an intermediate representation language. We develop a complete grammar for Nexus and we adopt this grammar to build a parser. 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 suitable for interconversion 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.

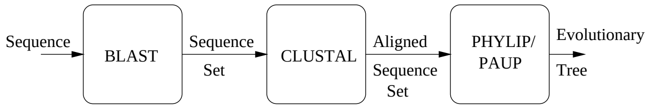

Fig. 1. A Standard Phylogenetic Inference Task Pipeline

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. These programs take input in a form not provided by the previous one and produce output incompatible with the next. Thus, between each of these scientifically relevant steps, translation from one format to another is required in order to accomplish the main goal. Unless the user is particularly adept at constructing shell scripts (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. Computational biologists are faced with a barrier when attempting to use available software tools: they must either devote significant energy in becoming programmers or they must spend endless hours editing and checking their data files when they convert from the output of one program to the input of another. 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 efforts of the BioPerl group [2]). Of these proposals, the Nexus data description language [10] represents the most comprehensive effort to provide a universal, computer oriented language for data description 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.

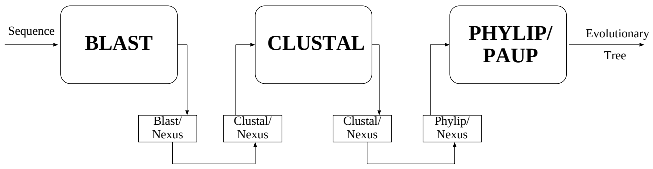

Fig. 2. Modified Pipeline—using Nexus as Intermediate Language

In the last few years Nexus has gained considerable success, but, in spite of this, the number of available tools to parse and manipulate Nexus files is 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 Nexus—including complex token definitions and a number of context-sensitive features—which makes the task of parsing arbitrary Nexus files 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 this important task has been accomplished. This technology is based on *logic programming (LP)* techniques—the parsing capabilities of Prolog, including *Definite Clause Grammars (DCGs)*, non-determinism, and the flexibility of unification and logical variables provide an excellent framework for tackling the complexity of Nexus (e.g., the context-sensitive components of the language).

This project represents the first step towards a more ambitious goal: the development of a *domain-specific language* [5] for describing biological processes at a high level of abstraction, and for creating reliable bioinformatics applications. In this paper we present the development of a specification of Nexus and its use to construct a complete parser. We also exhibit a preliminary approach for inter-conversion between Nexus and the PHYLIP [3] format.

2 Data Formats in Bioinformatics Applications

The abundance of 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 output forms depending on the the actual BLAST program (BLASTP, BLASTN, etc.) used and the choice of parameters for the search. E.g., 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 [7] tries to automatically recognize 7 input formats for sequences (e.g., `NBRF/PIR`, `EMBL/SWISSPROT`, `Pearson (Fasta)`, `CLUSTAL`). 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 and 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 program as input to another program. 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 PAUP or PHYLIP (unless the PHYLIP option was chosen). 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 [10]). 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.

Additionally, the fast development of the Internet has prompted the development of *markup languages* adequate to describe classes of biological data—e.g., BioML [14] and BlastXML [19]. This introduces the additional need of interconverting between these markup languages (currently used for data exchange purposes) and the more traditional formats recognized by existing applications.

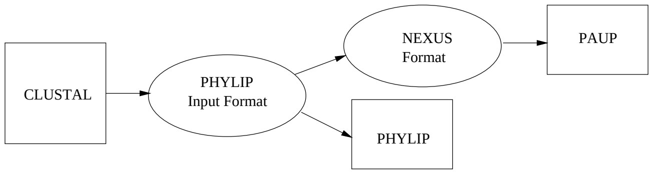

Fig. 3. Avoiding Wasteful Computation via Translation

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 data format design for most bioinformatics 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 in English. The Nexus data description language perhaps is the most well-documented format, but, even for Nexus, a formal description is not available. A formal description of the valid inputs and outputs for the commonly used bioinformatics tools, possibly via formal grammars, will go a long way towards format standardization and ease translation between formats.

3 Approach to Formats Interoperability

To achieve our goal, the methodology that we use should be general enough so that the task of format translation/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, we chose Nexus to be the common intermediate language [10]. Nexus represents a good starting point since it has been designed as a data description language, independent of any specific operating environment or application, and general enough to encompass the most relevant data types used in bioinformatics applications. Additionally, 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 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) to cover these missing capabilities.

Construction of the Translators: Most of the existing applications which use the Nexus format rely on *ad hoc* translators, which are used to extract and translate small subsets of the input file. 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 repeated effort, as the commonality among different translators is not exploited. General parsing of Nexus is a hard task. Two approaches can be considered to tackle this problem: *(i)* Manually develop a parser from scratch. This is a time consuming task and may produce unreliable parsers. Resulting parsers are always hard to verify. *(ii)* Use available tools to automatically generate parsers from grammars (e.g., YACC). However, YACC can handle only restricted types of languages (those for which a *LALR(1)* grammar is available). Thus, developing a complete parser using YACC is still a non-trivial task; there are various problems that one might encounter, including the fact that an LALR(1) grammar for the given language may be hard or impossible to construct. Of these approaches, the second one is considerably less complex, more reliable and theoretically better grounded. However, the issues discussed above indicate that building a parser using YACC for a non-trivial language is still a difficult task, outside the domain of expertise of most biologists. Furthermore, while neither of these approaches can be generalized to accommodate additional translators or language extensions (they are 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.

Logic Programming Technology: In this project we propose to use *logic programming (LP)* [16], 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. 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). On the other hand, the *Definite Clause Grammar (DCG)* facility of Prolog allows one to rapidly specify and implement parsers. 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. The use of DCGs, as mentioned, allows us to combine the process of creating a formal description of Nexus with the process of deriving a working parser. In addition, Prolog's non-determinism allows one to produce (from a DCG 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. Furthermore, since the non-terminals in a DCG parser are predicates, it is possible to add *arguments* to the components of the grammar allows us to create "communication channels" between the grammar elements, allowing to handle with ease the context-sensitive components of Nexus. Finally, the use of unification 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 format translation, it is possible to develop a single procedure which translates between two formats and use it to perform translation in both directions. Although this is difficult to achieve in general, the reversibility has been used to avoid rebuilding parts of the translation process.

In order to systematically develop our framework, we have taken inspiration from the work on *Horn Logic Denotational Semantics* [4]. 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. 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 System implementation

Description of Nexus: Nexus [10] is a data description language created 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 [10]: *(i) processibility:* unlike other formats, the design of Nexus promotes (semi-)formal specification to facilitate computer processing of Nexus files; *(ii) expandability:* the structure of Nexus files is modular, allowing the addition of new types of data blocks without disrupting the rest of the file structure; *(iii) inclusivity:* the choice of basic data types provided in Nexus is expected to cover all the needs of typical phylogenetic bioinformatics applications; *(iv) portability:* while most of the existing formats have been designed to fit the needs of a specific application, Nexus is an application independent language; its specification accounts also for operability within different computing environments. Nexus has gained considerable popularity in recent years. A number of applications (e.g., PAUP 3, MacClade, COMPONENT) have adopted Nexus as their data format. Nexus has also been adopted as a format for data housing in various public databases (e.g., TreeBase, Genetic Data Analysis).

A Nexus file, taken from [6], is shown in Fig. 5. 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, comes right after the `begin` word. For instance, in Fig. 5 there are four blocks, namely `data`, `codons`, `assumptions`, and `paup`. 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 [11]. Nesting blocks inside another Nexus block is not allowed. Each block is composed by several *commands* and each command may have several options. Each command is used to specify a certain property of the collection of data described by the block. In Figure 5 `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 [10]. 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 [10] is not based on formal 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 also led to a language containing 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 found in programming languages. These are discussed in the rest of this section. Moreover the Nexus standard is quite large. To meet 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 [10] with information drawn from user manuals of applications which use Nexus (e.g., PAUP) and data files obtained from the Web and from local biologists. Our BNF grammar definition of the Nexus standard [10] 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 [11]. Due to the size of the grammar, we show here just a small fragment of the BNF notation used to describe Nexus—Fig. 4 shows the part of the definition for the main Nexus structure. Note the similarity with the structure of a DCG. A complete description of the Nexus grammar can be found at `www.cs.nmsu.edu/~epontell/nexus`.

```
nexus  -->   "#NEXUS" blocks .
blocks -->   block blocks     |     .
block  -->   "begin" block_declaration  ";"  end  .
end    -->   "end"
       |     "endblock"  .
block_declaration -->
             block_taxa
       |     block_characters
       |     block_unaligned
       |     block_distances
       |     block_data
       |     block_codons
       |     block_sets
       |     block_assumptions
       |     block_trees
       |     block_notes
       |     block_unknown  .
```

Fig. 4. Nexus BNF fragment corresponding to the main format structure.

In order to translate from Nexus to another format, our program follows the traditional four phases (reading input, lexical analysis, parsing, and translation). The system reads the input file in a list of character codes. This list is then scanned creating a list of tokens. After this, the list of tokens is parsed using a *Definite Clause Grammar* (DCG) to create a parse tree which is suitable for translating to other formats. We shall devote the remaining of this section to describe these phases, emphasizing the implementation techniques adopted.

```
#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;
```

Fig. 5. A Sample Nexus File

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 lose some relevant information. It is worthwhile to mention that the format of a comment-command is very general and particular to each program. Thus, 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 a sequence of characters delimited by spaces or punctuation. However, there are some exceptions: a token can include spaces and punctuation if enclosed in quotation, and comments do not break tokens. E.g.,

`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 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 is interpreted as a blank space. Therefore, `DROSOPHILA_ASPARTIC_ACID` and the token in (1) represent the same syntactic object.

Another interesting property of the Nexus file format is that an identifier can start with a number. The scanner should be able to distinguish such identifiers from numbers (that can be either integers or reals). 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 Nexus and most tokens have different meaning depending on the 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 an identifier. Similarly, `2.34` in a data matrix represents either the value of a taxon frequency, or the symbols `2 . 3 4` corresponding to a DNA sequence. Besides, some single characters, like the newline, may or may not have a significant syntactic meaning depending on the context or according to several selected options in the file.

Due to the characteristics of Nexus, just a few token recognition decisions can be made during the scanning process, while most of them have to be delayed to the parsing phase. Our approach 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 6. The figure shows a formatted fragment of the list of syntactic elements produced by the scanner taking the file in Figure 5 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/character coordinates of the first character of the token in the file. The token coordinates are used during the parsing phase to communicate possible warnings, errors, or other messages to the user.

```
[  ['#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],    ...  ]
```

Fig. 6. Result of Lexical Analysis

Note that in Figure 6 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 combined in the parsing or the translation phase when this is required. Related to the issue of handling comments, Nexus allows special types 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*.

Parsing: Once a list of syntactic elements is created in the scanning phase, parsing is performed using Prolog's DCGs. The creation of the basic parser is relatively simple—as it can be obtained by translating the rule of the grammar specification of Nexus (e.g., the rules in Fig. 4) to DCG rules, in a straightforward manner. 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 original Nexus formats, the current standard [10] encourages support of every version of Nexus if possible. Further enhancements are also 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 new user-defined blocks, or even the introduction of new user-defined commands inside a standard block. In this regard, the standard [10] 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 that makes it impossible to recognize the entire file. To implement this kind of behavior, it is essential to rely on the use of non-determinism—letting the parser try the possible alternatives without commitment until a satisfactory one is found (if any).

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:

```
Begin Data;
    DIMENSION NCHAR = 7;
    FORMAT DATATYPE = DNA MATCHAR .;
    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 *directly* coded as a YACC rule (unless we use complex functions to perform replacements of symbols). 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. Additionally, 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 to the user.

There is a final important implementation detail in the parser. A DCG assumes that 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 (as seen previously), and unification does not work any longer as a token matching mechanism, due to the matching requirements described earlier.

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.          (1)
```

In our approach, the normal terminal symbol definition is replaced 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 (1) 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 data is usually stored in the parse tree and used in the translation phase to facilitate the conversion process, report errors, or handle the comments according to the target format specification.

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 according to the strategy presented in [5, 4]. 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 the previous section. 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.

5 Discussion and Related Work

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 the creation of a grammar for Nexus. The description given in the standard [10] is not formal at all—that may be one reason of why there is no other fully compliant Nexus parser (to the best of our knowledge). Our original BNF description of Nexus was intentionally done using a syntactical form very close to a DCG. Therefore, once we constructed the BNF we essentially had the parser completed. The only modifications to the code with respect to the BNF description is related to the special match routine and the extra argument 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. Furthermore, the processing of Nexus code requires the use of a number of data structures (e.g., lists, trees) which are readily available in Prolog.

With respect to the scanner generator issue, it is necessary to mention that there also exist publicly available scanner generators for Prolog, e.g., Elex(Prolog) and PLEX [15, 18]. 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) is quite flexible and has 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 LP community would benefit from the development of an optimized-scanner Prolog generator like LEX, especially knowing that compilers 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.

Finally, as a note in favor of Prolog, we should say that given the computing power of today's computers, it is 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 about 15 seconds on a standard PC.

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.

The work that comes closest to ours is represented by *NCL* [9], 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* [12] 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 LP as a technology to enhance interoperability between applications has recently gained relevance in a number of domains. Several examples have been developed in the context of Web formats [13] and in the conversion between formats for mathematics [8].

6 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 Nexus. 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 contains 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 an intermediate language to provide interoperability between different data 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 their support. This work is supported by a grant from the National Biotechnology Information Facility, 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. *Mol. Biol. Evol.*, 2:526–538, 1985.
2. BioPerl. *BioPerl Project: XML.* `bio.perl.org/Projects/XML`.
3. J. Felsenstein. PHYLIP: Phylogeny inference package, version 3.5c, 1993.
4. G. Gupta. Horn Logic Denotations and Their Applications. In *The Logic Programming Paradigm: The next 25 years*, Springer Verlag, May 1999, pp. 127-160.
5. G. Gupta and E. Pontelli. A Horn logic denotational framework for specification, implementation, and verification of domain specific languages. T.R., NMSU, 1999.
6. K. Hayasaka, T. Gojobori, and S. Horai. Molecular phylogeny and evolution of primate mitochondrial DNA. *Mol. Biol. Evol*, 5:626–644, 1988.
7. D. G. Higgins, J. D. Thompson, and T. J. Gibson. Using CLUSTAL for multiple sequence alignments. *Methods in Enzymology*, 266:383–402, 1996.
8. A. Karshmer, G. Gupta, S. Geiger, C. Weaver. Reading and Writing Mathematics: the MAVIS Project. *Behavior and Information Technology*, 18(1), 1999.
9. P.O. Lewis. NEXUS Class Library. `lewis.eeb.uconn.edu/lewishome/ncl2`.
10. 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.
11. W.P. Maddison and D.R. Maddison. *MacClade: Analysis of phylogeny and character evolution, version 3.07.* Sinauer, Sunderland, Massachusetts, 1997.
12. R. Page. A NEXUS Data Editor for Windows. `taxonomy.zoology.gla.ac.uk/rod`.
13. E. Pontelli, G. Gupta, et al. A Domain Specific Language for Non-Visual Browsing of Complex HTML Structures. *ACM Conf. on Assistive Technologies*, 2000.
14. Proteometrics. The BIOpolymer Markup Language (BIOML). `www.bioml.com`.
15. P. Reintjes. MULTI/PLEX: An AI tool for formal languages. Unpubl., 1994.
16. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1996.
17. D. L. Swofford. PAUP: phylogenetic analysis using parsimony, version 3.0. Technical report, Illinois Natural History Survey, 1989.
18. G. van Noord. Prolog(Elex): A tool to generate Prolog tokenizers. Unpubl., 1997.
19. WorkingObjects.com BlastXML. `workingobjects.com/blastxml`.