Semantics-based Filtering: Logic Programming's Killer App[1]

G. Gupta,[2] *H-F Guo,*[3] *A. Karshmer*[4]
E. Pontelli,[5]*, J. R. Iglesias,*[5] *D. Ranjan,*[5] *B. Milligan*[6]
N. Datta,[7] *O. El Khatib,*[5] *M. Noamany*[5]

Abstract:
We present a logic programming based framework for rapidly translating one formal notation $\mathcal{L}_s$ to another formal notation $\mathcal{L}_t$. The framework is based on Horn logical semantics—a logic programming encoding of formal semantics. A Horn logical semantics of the language $\mathcal{L}_s$ is constructed which employs the parse trees of the language $\mathcal{L}_t$ as semantic domains for expressing the meaning of sentences in $\mathcal{L}_s$. This formal semantics, coded in logic programming, immediately yields an executable (reversible) filter. This (reversible) filter is provably correct, as it is generated from the semantic specification. Our approach provides a formal basis for interoperability and is illustrated through four major practical applications: Translating Nemeth Math Braille notation to LaTeX, translating HTML to VoiceXML to make web-pages accessible via an audio-browser or a phone, translating ODBC programs/data to OQL (Object Query Language) programs/data, and interoperating between various biological software systems developed for phylogenetic inference via the NEXUS data representation language.

1 Introduction

The need to translate one formal notation to another formal notation arises very frequently. There are primarily two types of situations where this is needed:

- When a user wants to *migrate* a program from one machine or system to another machine or system. For example, migrating a program written in C from Solaris Unix to Linux or migrating a program developed for Linux to Windows NT. We refer to this migration as *porting*.
- When a user wants to translate a program written in one particular notation to accomplish a particular task to another notation that accomplishes the same task. For example, translating programs written in Fortran to equivalent programs in C, or program written for Oracle relational database to IBM DB2 relational database, or translating the internal representation of documents written in Word Perfect to that in Microsoft Word. We refer to this translation as *filtering*.

Both porting and filtering are manifestation of the same problem, in which we want to translate the solution to a problem expressed in one formal notation to another. We use the term porting to describe those situations where the changes needed in the notation are minimal (e.g., lexical tools may be enough to accomplish porting). For instance, while porting a C program from Solaris to Windows NT, code involving systems calls may have to be changed as systems calls may have different names/meanings/functions under different operating systems. Additionally, the differences in the two compilers involved may have to be taken into account. We use the term filtering, in contrast, to describe those situations where the notation in which the solution is expressed may have to be completely changed. The new notation may have entirely different syntax and its constructs may have widely differing semantics. Thus, lexical techniques are not enough, and the structure of the notation to

[1]Authors are partially supported by NSF grants CCR 99-00320, CCR 98-20852, CDA-9729848, HRD 9800209, EIA 98-10732, EIA 9729848, INT 9904063, and grant H133G010046 from the US Dept. of Education.
[2]Dept. of Computer Science, UT Dallas, Richardson, TX, gupta@utdallas.edu (contact author)
[3]Dept. of Computer Science, SUNY Stony Brook, Stony Brook, NY; haifeng@cs.sunysb.edu
[4]National Science Foundation and University of S. Florida, Tampa, FL; arthur@csee.usf.edu.
[5]Dept. of Computer Science, New Mexico State University, Las Cruces, NM; epontell@cs.nmsu.edu
[6]Dept. of Biology, New Mexico State University, Las Cruces, NM; brook@biology.nmsu.edu.
[7]Synopsis, Inc., Boston, MA.

be translated has to be inferred via parsing and semantic processing performed. Of course, a filter can be built only if it is possible to map the semantics of the source notation to (a subset of) the target notation. In this paper, we will only concern ourselves with the filtering problems, though the framework developed is trivially applicable to porting problems as well.

Suppose we wish to translate a sentence S (a program, a document, etc.) written in a language $\mathcal{L}_s$ to an "equivalent" sentence of $\mathcal{L}_t$. This would first entail building a general translator from $\mathcal{L}_s$ to $\mathcal{L}_t$, and then using this translator to translate S. The translator essentially is a map from semantics of $\mathcal{L}_s$ to the semantics of $\mathcal{L}_t$. In this paper we present an approach based on logic programming and formal semantics for rapidly developing this translator. In our approach the syntax and semantics of $\mathcal{L}_s$ is denotationally specified using Horn logic. The *meaning space* used for specifying the semantics of $\mathcal{L}_s$ consists of parse trees of $\mathcal{L}_t$. This denotational specification is executable and *automatically yields a translator*. This translator can be then used to translate sentences of $\mathcal{L}_s$ to equivalent sentences of $\mathcal{L}_t$. Because the translator is automatically generated from a semantic specification, it is provably correct.

Traditionally, language filters have been developed using standard compiler technology [1]. However, compared to the traditional approach our semantics based filtering framework has a number of advantages:

- If the syntax and semantics are specified with care, then the reverse translator can also be automatically obtained (i.e., a translator for translating sentences of $\mathcal{L}_t$ to equivalent sentences of $\mathcal{L}_s$ is obtained for free). This is, of course, due to relational nature of logic programs.

- Traditionally, porting and to some extent filtering have been done manually. Manual porting and/or filtering can be quite prone to error. However, using our approach, both the forward translator as well as the reverse translator obtained are provably correct as they are automatically obtained from the declarative semantic specifications. Further confidence can be gained by proving the correctness of the semantic specification: the semantic specification consists of (denotationally specified) recursive rules whose correctness can be manually established through inductive proofs.

- Translators for more complex languages, such as those that are context sensitive, can be elegantly specified using our approach. It should be noted that formal notations designed by non-computer scientists tend to be context sensitive. Traditional compiling technology is limited to parsing of context free, LALR(1) languages and is not designed to handle such context sensitive languages. Attempting to handle such languages using traditional compiling technology results in a very complex back-end, that is likely to contain errors, and that is hard to formally prove correct.

- A filter can be obtained in less time using our Horn logic semantics-based approach. Despite the availability of tools such as parser generators, etc., building a language filtering system is still a daunting task. Part of the difficulty comes from the fact that the semantic phase (the back-end) of this compiler is written in an imperative language such as C (e.g., if one uses YACC). In contrast, the syntax and semantic specification in our case is declaratively specified using logic programming.

An approach based on logic programming is ideally suited for translating formal languages, as historically speaking language translation was one of the two fields from which logic programming emerged (theorem proving being the other).

Porting and filtering have always been considered important problems in computing, especially in world of business computing, since the underlying software, operating systems, hardware, etc. change very frequently and require the user programs to be modified accordingly as well. It has acquired greater prominence with the advent of XML [7], which gives users the ability to define their own markup languages. Rapidly constructing translators or filters between one DTD (XML grammar specification) to another will become very important [21], as the number of DTDs designed for the same task proliferates. We believe that our logic programming-based technique will prove beneficial for such situations [10, 14]. The main advantage gained by using our Horn logic semantics based approach is that *reversible* filters can be built (approaches espoused by W3C for translating DTDs, such as XSLT transformation language, are good only for one way translation).

We believe that semantic filtering has the potential to become logic programming's killer application. The reversibility of the translation system, the speed with which it can be built and modified, its verifiable and

provably correct nature are some of the reasons behind this belief. Our experience in successfully applying this technology to widely varying situations, described in this paper, serves as empirical evidence.

2 Semantics-based Translation

2.1 Denotational Semantics

Denotational Semantics [28, 16, 29] is a well-established methodology for the design, description, and analysis of programming languages. In the denotational semantics approach, the semantics of a programming language/notation is specified in terms of mathematical entities, such as sets and functions. The denotational semantics of a language $\mathcal{L}$ has three components:

- *syntax specification:* maps sentences of $\mathcal{L}$ to parse trees; it is commonly specified as a grammar in the BNF format.
- *semantic algebra:* represents the mathematical objects used for expressing the meaning of a program written in the language $\mathcal{L}$; these mathematical objects typically are sets or domains (partially ordered sets, lattices, etc.) along with associated operations to manipulate the elements of the sets.
- *valuation functions:* these are functions mapping parse trees to elements of the semantic algebras.

Given the denotational semantics of a language $\mathcal{L}$, and a program $\mathcal{P}_{\mathcal{L}}$ written in $\mathcal{L}$, the denotation of $\mathcal{P}_{\mathcal{L}}$ w.r.t. the denotational semantics can be obtained by applying the top-level valuation function to the parse tree of $\mathcal{P}_{\mathcal{L}}$. The denotation of $\mathcal{P}_{\mathcal{L}}$ is an entity that is amenable to formal mathematical processing, and thus has a number of applications. For example, it can be used to prove properties of $\mathcal{P}_{\mathcal{L}}$, or it can be transformed to obtain other representations of $\mathcal{P}_{\mathcal{L}}$ (e.g., a compiled representation that can be executed more efficiently [11]). In this paper we assume that the reader is familiar with formal semantics. A detailed exposition can be found in [28, 29].

2.2 Horn Logical Semantics

Traditional denotational definitions express syntax as BNF grammars, and the semantic algebras and valuation functions using λ-calculus. The Horn Logical Semantics of a language uses Horn-clause logic (or pure Prolog) to code all the components of the denotational semantics of a language [11]. This simple change in the notation for expressing denotational semantics, while resulting in loss of some declarative purity, leads to a number of applications [11]. There are two major advantages that this simple change brings about:

1. A parser can be obtained from the syntax specification with negligible effort: the BNF specification of a language $\mathcal{L}$ can be trivially translated to a *Definite Clause Grammar* (DCG) [30]. This syntax specification, coded as a DCG, can be loaded in a Prolog system, and a parser automatically obtained. This parser can be used to parse programs written in $\mathcal{L}$ and obtain their parse trees. The semantic algebra and valuation functions can also be coded in Horn clause logic. As a result, both the syntax and semantic specifications are executable on any standard logic programming system. What is noteworthy is that different operational models will be obtained both for syntax checking and semantic evaluation by employing different execution strategies during logic program execution. For example, in the syntax phase, if left-to-right, Prolog style, execution is used, then recursive descent parsing is obtained. On the contrary, if a *tabling-based* [3] execution strategy is used then chart parsing is obtained. Likewise, by using different evaluation rules for evaluating the semantic functions, strict evaluation, non-strict evaluation, etc. can be obtained. Thus, interpreters and compilers (via partial evaluation of the interpreter) can also be obtained easily [11]. By using bottom-up [6] or tabled [3] evaluation, the fixpoint of a program's denotation can be computed.

2. Horn logical semantics can be used for automatic verification: the declarative nature of Horn logic can be used to verify interesting properties of programs. The Horn logical semantic of a language $\mathcal{L}_s$ can be viewed as an axiomatization of the language constructs of $\mathcal{L}_s$. The Horn logical meaning (denotation) of a program P written in language $\mathcal{L}_s$ can be thought of as an axiomatization of the logic implicit in the

program P or as an axiomatization of the problem that P is supposed to solve. This axiomatization can be used in conjunction with a logic programming engine (extended with negation) or a theorem prover to perform verification [11]. Additionally, as noted earlier, the relational nature of logic programming allows for the state space of a program (i.e., all possible execution paths of the program) written in $\mathcal{L}_s$ to be explored with ease, or to compute the fixpoint of the program's denotation which can be subsequently used for verification and structured debugging. The verification aspect of Horn logical semantics becomes more prominent when Horn logic is generalized to constraints for expressing semantics of languages as it allows for verification of real-time systems & languages [12].

Since both the syntax and semantics of the specification are expressed as logic programs, they are both executable. These syntax and semantic specifications can be loaded in a logic programming system and executed, given a program written in $\mathcal{L}_s$. This provides us with an interpreter that computes the semantics of programs written in the language $\mathcal{L}_s$. If the semantic domain used is that of a memory store, a regular interpreter is obtained, if the semantic domain used is constructs of another language $\mathcal{L}_t$, a translation of the program to the language $\mathcal{L}_t$ is obtained, etc.

2.3 Semantics-based Language Filtering

Horn logical semantics also provides a formal basis for *porting* or *language filtering*. Specification of a filter can be seen as an exercise in semantics. Essentially, the meaning or semantics of the language $\mathcal{L}_s$ can be given in terms of the constructs of the language $\mathcal{L}_t$. This meaning consists of both syntax and semantic specifications. If these syntax and semantic specifications are executable, then the specification itself acts as a translation system, providing a provably correct filter. The task of specifying the filter from $\mathcal{L}_s$ to $\mathcal{L}_t$ consists of specifying the definite clause grammar (DCG) for $\mathcal{L}_s$ and $\mathcal{L}_t$ and the appropriate valuation predicates which essentially relate (map) parse tree patterns of $\mathcal{L}_s$ to parse tree patterns of $\mathcal{L}_t$. Let $\mathcal{P}_s(S_s, T_s)$ be the top level predicate for the DCG of $\mathcal{L}_s$ that takes a sentence S_s of $\mathcal{L}_s$, parses it and produces the parse tree T_s for it. Let $\mathcal{P}_t(S_t, T_t)$ be the top level predicate for the DCG of $\mathcal{L}_t$ that takes a sentence S_t of $\mathcal{L}_t$, parses it and produces the parse tree T_t for it. Let $\mathcal{M}_{st}(T_s, T_t)$ be the top level valuation predicate that relates parse trees of $\mathcal{L}_s$ and $\mathcal{L}_t$. Then the relation

$$\texttt{translate}(S_s, S_t) \texttt{ :- } \mathcal{P}_s(S_s, T_s),\ \mathcal{M}_{st}(T_s, T_t),\ \mathcal{P}_t(S_t, T_t).$$

declaratively specifies the equivalence of the source and target sentence under the semantic mapping given. The `translate` predicate can be used for obtaining S_t given S_s and *vice versa*. Note, however, that $\mathcal{P}_s, \mathcal{P}_t$ and $\mathcal{M}_{st}$ have to be specified with care for the `translate/2` predicate to be truly reversible under Prolog.

Note that because the semantics is specified denotationally, the syntax and semantics specification of a language are guided by its BNF grammar: there is one syntax rule in the syntax specification per BNF production and one semantics rule in the semantics specification per BNF production. Occasionally, however, some auxiliary predicates may be needed while defining valuation predicates for semantics. Also, extra syntax rules may be introduced if the BNF grammar has left recursive rules which need to be eliminated in the DCG (by converting to right-recursive rules).

3 Applications

We next illustrate our framework with four large applications to which our approach has been successfully applied. Note that the application of our approach to the four cases was motivated by practical needs, where the user of our system was only interested in a one-way translation, and reversibility of the translation process was not an important factor. Thus, the source language parse trees were mapped directly to phrases of the target language (i.e., the parser $\mathcal{P}_t$ was only implicitly specified). In all 4 cases, the translator was specified in record time, to great surprise of the users whose problem we were solving.

3.1 Translating Nemeth Math Braille Code to LaTeX

We first consider the problem of translating Nemeth Math Braille documents to LaTeX. This project was done for the MAVIS (Mathematics Accessible to Visually Impaired Students) group at New Mexico State University

and funded by the National Science Foundation. The MAVIS group was set up by NMSU to assist blind students in their study of Mathematics and Science. The goal of the project was to make Mathematics more accessible to visually impaired students and scholars of Mathematics. Nemeth Math Braille is a notation used for encoding Mathematics much in the fashion of LaTeX. It was designed for two main reasons: (i) normal Braille permits only 64 characters (as it is six dots based) and is inadequate for encoding mathematics; and, (ii) to convey the structure of mathematical formulas to blind students. Nemeth Math Braille code was designed in 1951. The rules of grammar are specified in English and are quite complex (the specification runs about 250 pages [25]). Many of the rules are context sensitive in nature, we believe, for three reasons: (i) since the notation is designed for the blind, an attempt is made to keep them aware of the context at all times, resulting in context sensitive features; (ii) the language was designed when very little was understood about grammars and languages; our experience indicates that when non-computer scientists design language, they almost always end up including context-sensitive features in the language; and, (iii) the Nemeth Math Braille Code was primarily designed to transcribe printed mathematics into Braille in a way that the relative spatial placement of symbols is preserved; this also adds to context sensitivity and makes parsing harder (in fact, many features of Nemeth Math Braille code preclude building an LALR(1) parser for it.) As a result of all these problems, automatic processing of Nemeth Math Braille code (so as to obtain an equivalent document in LaTeX) was considered an insolvable problem by researchers working on developing assistive technology for the visually impaired [24, 27].

The translation to LaTeX is needed to facilitate communication between a sighted course instructor and a blind student: the sighted instructor gives a homework written in LaTeX, which is automatically translated to Nemeth Math Braille code (such a translator has already been built using traditional compiler technology by the MAVIS group, as LaTeX has an easy context free grammar); the blind student "reads" the Nemeth Math Braille coded document, and answers it in Nemeth Math Braille. However, the sighted instructor will not be able to check the answers (since most likely he/she cannot understand Nemeth Math Braille code) unless a translation of the answers to LaTeX is performed.

Nemeth Math Braille code is a complex, context sensitive language, specified informally via examples [25] (first specified in 1951, revised in 1972). An example of context sensitivity found in Nemeth Math Braille code is in the coding of superscripts and subscripts: the mathematical expression $x^a + b$ is represented by the Nemeth Math Braille code as xˆa"+b (we use ASCII equivalents instead of writing Braille code for x, ˆ, a, ", +, and b). However, another mathematical expression y^{x^a+b} is represented by Nemeth Math Braille code as yˆxˆˆaˆ+b, where ˆ is superscript indicator, ˆˆ means second order superscript indicator, and " is the base line indicator. It is easy to see that the same subexpression $x^a + b$ is represented as xˆˆaˆ+b rather than xˆa"+b because of the context environment of superscripts. Since the information about the context environment is also encoded in the Nemeth Math Braille code, this context information has to be analyzed during the parsing procedure so that the correct syntax structure will be generated. This makes the grammar of Nemeth Math Braille code very complex, making the syntax hard to specify. Note that several of the context sensitive features (including the example above), can be parsed by first writing a context free grammar that accepts a larger language, and then ruling out the illegal sentences during the semantic phase. However, in case of Nemeth Math Braille code this approach can only be followed in certain cases, as, otherwise, it results in a very complex semantic specification.

Fortunately, Definite Clause Grammars (DCG) of logic Programming can be used for encoding and obtaining a parser for context-sensitive grammars as well. DCG can recover the context information from the source language and then combine it to produce the parse tree. Thus, following our Horn logical semantics framework, we first construct a definite clause grammar for Nemeth Math Braille code, followed by the semantic mappings from Nemeth Math Braille code parse trees to LaTeX. In the semantic specification, the mapped values of some of the terms are dependent on the information in the context environment. That is, the same syntax term (parse tree pattern) will be mapped to different semantics depending on the context information it occurs in. In order to obtain the correct semantics, we have to make the context information visible to the whole or partial sentence being transformed.

Once again we present a small fragment of the code below which handles the translation of *polynomials* (these polynomials allow other polynomials to occur as exponents). We first give the syntax of such polynomials (expressed in Nemeth Math Braille code) as a DCG that does the parsing and produces the parse tree (for simplicity we restrict the number of variable names allowed). Then we give the semantics that translates parse

trees produced to the corresponding LaTeX mathematics expressions. Since the LaTeX to Nemeth Math Braille code translator already existed at that time, reversibility of the system was not a concern. Thus, the semantics of Nemeth Math Braille code was given directly as LaTeX sentences rather than as LaTeX parse trees.

```
% Definite Clause Grammars
exp(e(X)) --> term(X).
exp(e(T, O, E)) -->
    term(T), op1(O), exp(E).
term(t(X)) --> vari(X).
term(t(V, H, T)) -->
    vari(V), hats(H), term(T).
op1(op(H, +)) --> hats(H), [+].
op1(op('"', +)) --> ['"'], [+].
op1(op(+)) --> [+].

hats([^]) --> [^].
hats([^|L]) --> [^], hats(L).

vari(x) --> [x].
vari(y) --> [y].
vari(a) --> [a].
vari(b) --> [b].
vari(c) --> [c].
```

```
% Semantic Specification
sexp(e(X), INL, L) :-
    sterm(X, INL, ONL, L1),
    getlist(ONL, Bl),
    append(L1, Bl, L).
sexp(e(T, O, E), NL, L) :-
    sterm(T, NL, OL, L1),
    sop(O, OL, NL1),
    Close_Braces is OL - NL1,
    getlist(Close_Braces, Bl),
    sexp(E, NL1, L2),
    append(L1, Bl, Lp),
    append(Lp, [+|L2], L).
sop(op(H, +), _, Ct) :- shats(H, Ct).
sop(op('"', +), _, 0).
sop(op(+), CL, CL).
shats([^], 1).
shats([^|L], Ct) :- shats(L, Ct1), Ct is Ct1 + 1.
sterm(t(X), OL, OL, [X]).
sterm(t(V, H, T), Ct, OL, [V, ^, '{'|R]) :-
    Ct1 is Ct + 1, shats(H, Ct1),
    sterm(T, Ct1, OL, R).
getlist(0, []).
getlist(N, ['}'|L]) :-
    N > 0, N1 is N - 1, getlist(N1, L).
```

The syntax and semantic specifications are executable, and thus automatically provide us with a translation system. Thus, if we load this program on a logic programming system and ask the following query:

```
?- exp(T, [x, ^, y, ^, ^, a, ^, +, b, '"', +, c], []), sexp(T, 0, L).
```

then the following answer is obtained:

```
T = e(t(x,[^],t(y,[^,^],t(a))),op([^],+),e(t(b),op('"',+),e(t(c))))
L = [x,^,{,y,^,{,a,},+,b,},+,c],
```

which corresponds to the LaTeX expression $x\hat{}\{y\hat{}a+b\}+c$, i.e., $x^{y^{a}+b}+c$.

Thus, the problem of designing a language translator [15, 11, 20] from Nemeth Math Braille code [25] to LaTeX can be easily solved in our formal framework. A translator from Nemeth Math Braille code to LaTeX was not available until we built one using our approach. In fact, a complete translator has been developed by our group. It took us only a 2-3 man months of work to produce the translator and to accomplish a task that was considered impossible by researchers in assistive technology [24, 27]. Our translator is being α-tested at the Texas School for the Blind and Visually Impaired and other sites. Eventually, it will be made available as an add-on to Scientific Notebook software (a WYSIWYG LaTeX based word-processing system) that already has the LaTeX to Nemeth Math Braille code translator built in. Our framework is currently being used, with funding from the US Dept. of Education, to develop filters between the Marburg notation (a Braille based notation for encoding Mathematics used in Europe) and LaTeX and the Nemeth Math notation so that blind and sighted students/scholars/researchers of Mathematics from US and Europe can communicate.

3.2 Interoperability Among Bioinformatics Software Systems

Next we look at the application of our translation framework to solving a practical problem in computational biology. The project was done for the National Biotechnology Information Facility (nbif.org), a University-Industry-Government consortium for promoting research in computational biology. The specific problem presented to us was to make various software tools developed for bioinformatics work smoothly with each other.

In the bioinformatics field, several software tools have been created for very specific tasks in order to make genetic information publicly available and to analyze it. A common bioinformatics analysis usually consists of the following steps:

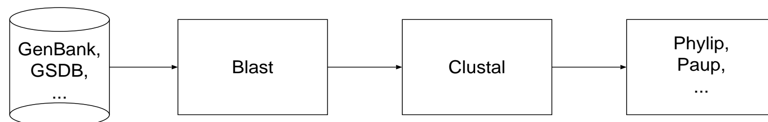

Figure 1: Ideal pipeline process to analyze a molecular sequence.

1. BLAST (Basic Local Alignment Search Tool) [2] is used to query a public database of genetic information like GenBank and GSDB. The result of this process is a molecular sequence.
2. A program like CLUSTAL W [18] is used to align the molecular sequences.
3. The molecular sequences is analyzed to infer phylogenetic information using programs such as PAUP [32], PHYLIP [5], or MACLADE [23].

These steps conform to a straightforward pipeline as shown in Fig. 1. Nevertheless each piece of software intervening in this process was developed independently and as a consequence many different (and complex) file formats are used for each program. For instance, PHYLIP can only accept a PHYLIP format file as input; the input of MACLADE and PAUP has to be specified in NEXUS [22] format; CLUSTAL W produces a specific CLUSTAL W format file as output, and so on.

This diversity of file formats clearly produces a compatibility problem between each of the stages in the pipeline of Fig. 1. To get around this problem, a biologist has to write ad-hoc scripts using tools such as `sed`, `awk` or `Perl` to extract the genetic sequences produced by one program and translate them into the format that can be used for the other programs, or do it manually. Manual translation is error prone and labor intensive. Writing of `Perl/awk/sed` scripts requires the biologist to have a reasonable computing background.

The approach currently taken to solve this problem by some software analysis tools is to make an effort to help by trying to recognize as many types of input formats as possible and trying to generate as many types of output formats as possible. For instance, CLUSTAL W tries to automatically recognize seven different input formats and to generate five different output formats. However, since the main concern of CLUSTAL W is sequence alignment and not sequence translation, the recognition procedure used in CLUSTAL W is not warranted to work all the times, the files generated by CLUSTAL W usually need some editing before they can be used by other programs, and some important file formats are not supported.

Our approach to this problem is to develop a bidirectional translator for inter-program communication using our Horn logical semantics approach [19]. In our design the output of each of the programs depicted in Figure 2 is taken by the translator and transformed into an internal representation. The desired conversion between the stages of the pipeline is achieved transparently through the transformation of the internal representation into the desired output format. The main advantage of using an internal representation is that for every format f we just need a bidirectional translation between f and the internal representation.

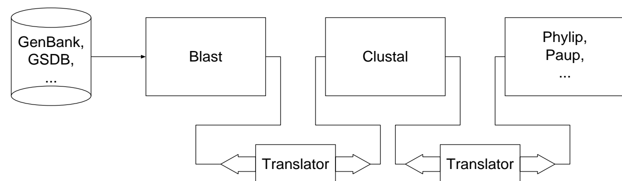

Figure 2: Proposed approach to achieve the pipeline process to analyze a molecular sequence.

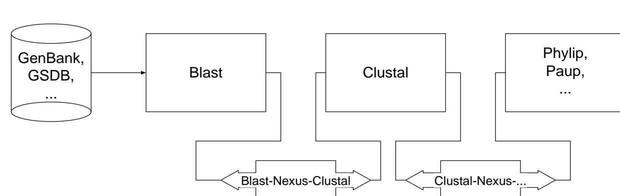

Figure 3: Implemented pipeline.

As illustrated in Figure 3, the internal representation chosen for the bidirectional translator is NEXUS [22], a modular and expandable format specially designed (by biologists) to comprehensively describe all genetic sequence information ideally needed by a systematic biologist. Even though many other formats exist to

represent this sequences, most of them are for a special purpose or contain information that is used by a particular program. On the contrary, NEXUS has gained attention in the last few years because of its interesting features and its goal of providing a uniform input and output format for all types of genetics related softwares. Each different kind of information in NEXUS is enclosed in a block which makes it very readable by a human. Another important concept is that the standard is very flexible allowing to define new kinds of blocks for hosting data for possible future biological applications. NEXUS files are portable since NEXUS is a format independent of any application, while it also allows to define information for a specific program encapsulating it inside a program-specific block. Currently in NEXUS it is possible to define all the structures required for systematic biology applications including, for instance, taxa, states, trees, matrices, unaligned data, distances, codons, and sets. We show a sample of a small NEXUS file below.

```
#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;
```

The tool for format interoperability we have developed has been constructed using the Horn logical semantics approach; Semantics of NEXUS is expressed in terms of the various formats to obtain reversible translators. The core of this approach has been implemented using Prolog (Prolog's built-in DCGs for syntax specification and regular Prolog for semantics specification). Describing the syntax/semantics of the various formats (in particular of NEXUS) has been accomplished following the traditional steps taken by a compiler. First, the

program reads the file into a list of character codes. Second, the codes are scanned and converted into a list of tokens. Third, the tokens are parsed using a DCG and a parse tree created. Finally, the parse tree is given an appropriate semantics.

However, the techniques used in each of these phases in this project are non-traditional because the internal format representation, i.e., NEXUS, is in fact a highly context sensitive language. The flexibility of NEXUS is a real implementation challenge and its DCG is very large. In fact NEXUS is in many ways more similar to a human language than to a usual computer language. In the remainder of this section we shall discuss some of the techniques used in the implementation of each of these phases. NEXUS underscores our point regarding language design: formal languages/notations designed by non-computer scientists tend to have context sensitive features.

Scanning: During scanning a list of character codes is tokenized in a list of syntactic elements with the following format: `[ Row, Column, Token ]`. The elements of a syntactic element represent the token recognized and the row and column coordinates of the token in the file. The coordinates are used in later translation stages to send messages to the user, e.g., warnings and errors. The scanning techniques used in the translator are quite unusual and heavily rely on non-determinism to handle the complexity of token definition in NEXUS. This was due to a number of reasons: (i) tokens in NEXUS are non-case sensitive; (ii) comments can occur inside tokens, (iii) Underscore maybe interpreted as blank space, (iv) comments can be nested and are of multiple types; comments can have impact on the meaning of a file (for example a command-comment `[&U]` in a tree block means that the tree is unrooted); (v) recognition of tokens and key words is sensitive to the context in which they appear.

The traditional DCG technique consists of creating in advance a list of tokens and using this list as a *difference list* in the DCG to parse the file. It is clear from the previous NEXUS description that it does not make too much sense to create such a list of tokens in advance since the real meaning of the tokens and its real character components are defined by the context. The technique we propose and follow consists in scanning and parsing simultaneously. Every time we reach a rule during parsing that looks for a token we call a predicate `match/4`, which scans the current input stream *guessing* to check whether the current token corresponds to the token expected. Whenever the input does not correspond to the expected token, `match/4` fails leaving the current input intact so that other DCG rule can be applied. This effectively allows multiple symbols look-ahead, which is indispensable for parsing NEXUS. Thus, YACC style LALR(1) parsing using traditional compiling technology will be too cumbersome to use.

The technique is briefly illustrated via a simplified rule fragment shown in Figure 4. The first two arguments of `match/4` are the token and the type expected. The third argument tells whether match should either guess or act deterministic. The last argument is the resulting syntactic element matched as described at the beginning of this section.

```
block_distances( [ N1, N2, N3, N4, N5, N6, N7, N8 ] ) -->
      match( 'distances', _, _, N1 ), match( ';', _, y, N2 ),
      dimensions_distances( N3 ),
      format_distances( N4 ),
      taxlabels_optional( N5 ),
      match( 'matrix', _, y, N6 ),
      matrix_data( N7 ), match( ';', _, y, N8 ).
```

Figure 4: Use of `match/4`

Parsing: The creation of a parser in Prolog is relatively simple using DCGs. However many factors had to be taken into account in this project due to the nature of the translated file formats and using NEXUS as internal file representation.

Since the system is allows translation between many different formats, modularity is important. A Prolog module is provided for each target format supported. Adding support for a new format in the future should imply just writing and *consulting* one more module for the corresponding format, without having to rewrite or modify the current program. We believe that this approach shall simplify future development and maintenance of the program.

In some other parsing tools, like YACC, one has to write the whole attribute grammar rules in one file. This is quite cumbersome, specially when we are dealing with huge grammars like NEXUS in which just the BNF grammar describing it consists of over five thousand lines, and the DCG with added semantic rules (which is the equivalent to a YACC attribute grammar) is around fourteen thousand lines long. Keeping a YACC file of such a length is clearly inappropriate. On the contrary, one can easily create different small Prolog file modules that can be easily maintained.

The files containing genetic sequences can occasionally be quite big, some of them in the order of several hundred kilobytes. Even though most Prolog implementations nowadays can easily read such a big file in memory representing it with a list of characters, the process of reading the entire file in advance for parsing using some traditional technique such as a *failure driven loop* might still be slow. The approach we proposed and successfully implemented consists in replacing the traditional difference list by a *stream stack*. Each element in the stream stack has the following format:

```
[ StreamPointer, Row, Column, PreviousCharacter ] .
```

The `StreamPointer` points to the current location of the file. The elements `Row` and `Column` indicate the coordinates of the character pointer indicated by `StreamPointer`. The `PreviousCharacter` is the character right before `StreamPointer` and it is used to keep Row and Column correctly updated independently of the current file system used (Unix, Mac, DOS, etc.)

At the very top of the stream stack the information corresponding to the current input stream to be parsed is located. Some elements in the stack can be pushed or popped at parsing time according to whether a file is included (for example by a `include` directive), or finished to parse (when the end of file is reached.)

In this manner, in the `match/4` procedure described previously, instead of *reading* a character code from a difference list, `match/4` calls `read_code/6` which reads tokens directly from the current input file.

Translation: Finally, in the translation stage the parse tree produced during parsing is taken and mapped to the required output format using the Horn logical approach described earlier.

Implementation: The actual implementation of the translator has been very satisfactory. It was produced in a very short time. Currently it supports bidirectional translation among NEXUS, Phylip and CLUSTAL W. These translators have been tested on user files provided by biologists; they work for files as big as several hundred kilobytes. More formats are being added and, as discussed previously, adding support for these formats consists of adding a Prolog module without modifying the entire program. In a short time these translating services will be fully available to the bioinformatics user community through the `nbif.org` web server.

The Horn logical semantics based approach greatly facilitated the implementation of this project. The complexity of the whole translator is reduced to specifying the BNF grammar, every thing else falls through automatically (since only one syntax rule and one semantic rule is needed per BNF production). Coming up with the grammar of each file format and NEXUS was the most time consuming task in this project because of the lack of a formal format definition in the literature. Once we had the grammar for each format, obtaining a DCG and specification of the valuation predicates was simple to achieve. Building these translators took less than 3 man months of work, most of which was devoted to formalizing the informally specified grammars of the various formats. In fact, the DCG grammar of NEXUS is the first ever complete parser ever built for NEXUS. Most parsers built to date are *ad hoc* parser for smaller subsets of NEXUS; complete parsers will be quite complex to build using traditional technology due to NEXUS' context sensitive and non-LALR(1) nature.

3.3 HTML to VoiceXML

The next semantic filtering problem we consider is translation of HTML to VoiceXML. HTML is the mark-up language widely-used to structure documents on the Web. VoiceXML is an XML based languages for marking up voice data. The goal behind VoiceXML is to make the Internet accessible through the phone or an *audio browser*. VoiceXML is being developed by a consortium of IBM, Lucent, Motorola and AT&T and requires a *voice browser* that understands VoiceXML and that reads out the information contained in the VoiceXML marked-up web page. VoiceXML also allows the user to interact (through speech) with the web-page via the audio browser or the phone.

For illustration purposes, we have only chosen a subset of the HTML containing the form construct. Since HTML has a well-formed context free grammars, it is very natural and easy to write the DCG grammar for this

subset as shown below, where almost one DCG rule corresponds to one BNF grammar rule. The building of the parse tree is also very easily specified, by adding an extra argument in which the parse tree is synthesized. The parse tree of a phrase is constructed in terms of the parse trees of its component sub-phrases. The DCG rules above constitute an executable program, as they will be translated into ordinary Prolog clauses automatically when they are loaded into the Prolog system. Thus, one can completely avoid having to deal with the algorithmic or computational aspects of parsing. Definite Clause Grammars allow context-free grammars to be easily expressed in Prolog, and the grammar specification automatically acts as a parser.

```
html_document(html_document(X)) -->  html_tag(X) .
html_tag(html_tag(X)) -->  ['<html>'], html_content(X), ['</html>'].
html_content(ht(X,Y)) -->  head_tag(X), body_tag(Y).

head_tag(head_tag(X)) -->  ['<head>'], title_tag(X), ['</head>'].
title_tag(title_tag(X)) -->  ['<title>'], plain_text(X), ['</title]>'].

body_tag(body_tag(X)) -->  ['<body>'], xbody_content(X),['</body>'].
xbody_content(xb(X,Y)) --> body_content(X), xbody_content(Y).
xbody_content(xb(X)) --> body_content(X).
body_content(X) --> text(X).
boby_content(X) --> form_tag(X).

form_tag(form_tag(X)) -->  ['<form>'], xform_content(X), ['</form>'].
xform_content(xf(X,Y)) --> form_content(X) ,xform_content(Y).
xform_content(xf(X)) --> form_content(X).

form_content(input_radio(X, Y)) --> plain_text(X), [:], xradio(Y).
form_content(select_option(X,Y)) --> [<],[select],[name],[=],string(X),[>],
    xoption(Y),['</select>'].

xradio(xradio(X, Y)) --> radio(X), xradion(Y).
xradio(X) --> radio(X).
radio(radio(X, Y, Z)) --> plain_text(X), [<],[input],[type],[=],[radio],
     [name],[=],string(Y),[value],[=], [#],string(Z),[#],[>].

xoption(xoption(X, Y)) --> option(X), xoption(X, Y).
xoption(X) --> option(X).
option(option(Y)) --> ['<option>'], plain_text(Y),['</option>'].
```

Specifying the semantics in terms of VoiceXML is the next step in the Horn logic semantics-based approach. Predicates that map HTML parse tree patterns to the corresponding sentences of the VoiceXML are specified. These predicates give a meaning to each term in the parse tree a value depending on the values of its subterms. In Prolog, we can write the valuation predicates as shown below:

```
to_voice(html_document(X), T) :- shtml(X, T1),
    append([['<?xml version=1.0>'], ['<vxml>'], ['<form>']], T1, T2),
    append(T2, [['</form>'], ['</vxml>']], T).
shtml(html_tag(X), T) :- sht(X, T).
sht(ht(H, B), T) :- shead(H, T1), sbody(B, T2), append(T1, T2, T).
shead(head_tag(X), T) :- shead_tag(X, T).
sbody(body_tag(X), T) :- sbody_tag(X, T).

shead_tag(head_content(X), [['<block> page Title:', T1, '</block>']]) :-
    shead_content(X, T1).
shead_content(title_tag(X), X).
sbody_tag(xb(X, Y), T) :- sxb(X, T1), sbody_tag(Y, T2), append(T1, T2, T).
sbody_tag(xb(X), T) :- sxb(X, T).

sxb(text(X), [['<block>', X, '</block>']]).
sxb(form_tag(X), T) :- sform_tag(X, T).
sform_tag(xf(X, Y), T) :- sxf(X, T1), sform_tag(Y, T2), append(T1, T2, T).
```

```
sform_tag(xf(X), T) :- sxf(X, T).

sxf(select_option(X, Y), [['<field name =', X, '>'], T3, T4, ['</field>']]) :-
    sxoption(Y, T1, T2), append(['<prompt>' | T1], ['</prompt>'], T3),
    append(['<grammar>' | T2], ['</grammar>'], T4).

sxf(input_radio(X, Y), T) :-
    sxradio(Y, T1, T2, N), append(['<prompt>' | T1], ['</prompt>'], T3),
    append(['<grammar>' | T2], ['</grammar>'], T4),
    append([['<block>', X, ':', '</block>'], ['<field name =', N]], [T3], T5),
    append(T5, [T4, ['</field>']], T).

sxoption(xoption(X, Y), T, TT) :- soption(X, T1), sxoption(Y, T2),
    append(T1, ['OR'|T2], T), append(T1, ['\|' | T2], TT).
sxoption(option(X), [X], [X]).

sxradio(radio(X, Y), T, TT, N) :-
    sradio(X, T1, N), sxradio(Y, T2, TT2, N),
    append(T1, ['OR'|T2], T), append(T1, ['\|' | TT2], TT).
sradio(radio(X, Y, _), [X], Y).
```

Note that the semantics is written denotationally, and that there is one semantic rule per grammar rule. Since we have defined the mappings of the valuation functions on all of the options listed in the DCG rules for the subset of HTML, the valuation functions are completely defined. The above syntax and semantics specification can be loaded on a Prolog interpreter and executed to translate sentences from HTML to VoiceXML. Thus, given the input HTML document to the left, an equivalent VoiceXML document shown to the right is obtained.

```
% HTML program
<html>
<head>
<title> Menu </title>
</head>
<body>
Menu
<form>
<select name=MenuApp>
<option> Chicken </option>
<option> Meat </option>
<option> Fish </option>
<option> Vegetable Salad </option>
</select>
</form>
<form>
Choose your desert : Ice Cream
<input type=radio name=omar value="icecream">
Apple Cake
<input type=radio name=omar value="applecake">
</form> </body> </html>
```

```
% VoiceXML
<?xml version=1.0>
<vxml>
<form>
<block> Page Title:  Menu  </block>
<block> Menu  </block>
<field name=MenuApp>
<prompt>  Vegetable Salad  OR  Fish  OR
          Meat  OR  Chicken </prompt>
<grammar>  Vegetable Salad  Fish  Meat
           Chicken </grammar>
</field>
<block> Choose your desert : </block>
<field name=omar>
<prompt> Ice Cream OR Apple Cake</prompt>
<grammar> Ice CreamApple Cake</grammar>
</field>
</form>
</vxml>
```

The main advantage of using Horn logical approach for converting HTML to VoiceXML was the speed with which the task of building the translator was accomplished. Also, for the subset of HTML considered for the project, the translator produced was reversible and could translate VoiceXML documents back to HTML. Note that an approach based on Horn logic can also naturally handle non-well formed HTML (i.e., ending tags may be omitted) through non-determinism in DCGs. Non-well formed HTML does not directly admit an LALR(1) grammar, for this reason parsers for HTML based on traditional compiler technology are large and quite complex [7]. Work is in progress to build a complete translator from HTML to VoiceXML.

3.4 Database Interoperability

In the final example, we consider the problem of making two databases interoperate with each other—one relational and another object-oriented. We consider the problem of translating progams and data expressed in a relational database format to an object-oriented database format. In particular we consider the translation of ODBC (a standardized SQL) programs and data to OQL (a standardized Object Query Language) programs and data. ODBC (Open DataBase Connectivity) was developed as a standard to make the development of relational database applications independent from the actual database environment the application was going to run on (Oracle, Sybase, Ingres, etc.). For each vendor's database environment, a driver has to be defined (by that vendor) that maps ODBC programs/data to the vendor's programs/data. The drivers for various relational systems are relative easy to define, as they map programs/data of ODBC, which can be thought of as a dialect of SQL, to other dialects of SQL. However, if we wish to make ODBC work with object-oriented databases, the driver becomes a lot more complex, as programs and data expressed relationally in ODBC have to be transformed to programs and data expressed in an object-oriented way. A solution to this problem is to build semantic filters that map ODBC programs/data to OQL program/data. OQL (Object Query Language) is a standard devised for query languages for object-oriented databases. The driver can thus be thought as a semantic filter from ODBC to OQL.

We adopt the usual Horn logical semantics approach to translation. In the syntax specification part the syntax of ODBC programs is specified as a *definite clause grammar* (DCG). This DCG is extended so as also to generate *parse trees* for the programs. In the semantic specification part, the semantics of ODBC constructs is denotationally given in terms of OQL constructs. Both the syntax and semantic specification are expressed using Horn logic. The Horn logic specification is executable and, thus, automatically yields a system that transforms ODBC programs to OQL programs.

The translation task is facilitated by the fact that the syntax of OQL is very heavily influenced by SQL. The main problem to be handled is mapping relational data representation (relational tables) to object data representation (persistent objects). In our semantic mapping, each table is mapped to a class, the columns are the attributes, and a row of a table is mapped to an instance of an object in that class.

A translator for the full ODBC language was developed, to show the applicability of Horn logic based semantics filtering technology to facilitate database interoperability [4].

Our logic programming based approach to database interoperability can also be used for making heterogeneous databases work together. Essentially, filters can be declaratively described between programs/data of each type of database involved, or the most powerful database (in terms of query expressiveness) identified and programs/data of all other databases semantically mapped to program/data of this database. Thus, Horn logical semantics arguably provides theoretical basis to interoperability among heterogeneous databases.

4 A Killer Application for LP

We strongly believe that semantic filtering can be a *killer application* for logic programming. Filtering and porting problems arise very frequently in business and industry. Rapidly solving these problems with the aid of logic programming and Horn logical semantics can lead to considerable productivity gains. These gains are evident from our experience working on building filters for software systems for bioinformatics and building the back-translator for Nemeth Math Braille code to LaTeX.

Also, the advent of XML creates a fertile area for the application of this technology, as the number of DTDs (document type definitions) proliferates. XML (eXtensible Markup Language) is a notation for specifying mark-up languages for a particular domain. The idea behind XML is to let groups of users define their own task-specific mark-up languages (i.e., their own set of tags with predefined meanings, and syntax rules that govern nesting of these tags, termed DTD) for marking up documents so that they can be further processed automatically. With a number of DTDs or tagging scheme being defined for the same task, one needs methods to convert documents marked-up using one DTD to another. One also needs to translate a DTD to HTML, so that XML documents can be displayed on the web. The WWW consortium has come up with XSL (eXtensible Stylesheet Language) and XSLT (XSL Transformations) for programming these translations. XSL/XSLT are still evolving and are not completely defined yet. We believe that language filtering technology outlined in

this paper is more effective in providing solutions to these transformation problem. The major advantage of a Horn logical based technology is that it allows transformations to be expressed in a reversible manner, that is, a translator specified for translating a DTD $D1$ to another DTD $D2$ automatically yields a translator for translating $D2$ to $D1$.

Logic programming based semantic filtering works in situations where other traditional approaches may not, for example, if the language is context sensitive or if it is not LALR(1). In such cases traditional compiler based remedies don't work, or produce a highly complex system. In case of the Nemeth Math Braille code to LaTeX translation problem, the complexity of the problem was such that it was deemed impossible to solve using traditional compiler technology. However, with our Horn logical semantics based approach we were not only able to solve the problem, we were able to build the system in record time. The same is true of our work on building translators for bioinformatics software systems.

At present we are using the framework presented in this paper to build a number of other systems, in particular, filters between the Marburg Braille notation, LaTeX, and Nemeth Math, resource (e.g., power, execution time) aware compilers for embedded systems, translator for SCR notation [17] to C, etc.

5 Related Work and Conclusions

Traditionally compiling technology has been used for solving filtering problems. However, as discussed earlier, traditional compiling technology is limited to context free, LALR(1) languages. Many languages designed by non-computer scientists tend to be context sensitive for which the traditional approach does not work.

The work that comes close to ours is that of Stepney [31] who uses a similar approach to declaratively specify compilers: the backend is specified by declaratively mapping parse tree patterns to machine instructions. Stepney's goal is to build compilers in a provably correct way rather than semantic filters.

In this paper we presented an approach based on logic programming and formal semantics for declaratively specifying translators. Our approach was illustrated via practical applications of this technology in four diverse areas: (i) making web accessible on the phone; (ii) making mathematics accessible to blind students; (iii) making biological software systems developed for phylogenetic inference interoperate with each other; and (iv) making relational databases work with object-oriented databases. Other applications of this framework are currently being investigated, namely, rapid interpretation and compilation of domain specific languages [13] and building XML parsers (a parser for an XML can be thought as a semantic map between its DTD rules stated in the EBNF notation and its corresponding DCG rules). A general tool for graphically specifying mappings between notations that will automatically generate the Horn logical syntax and semantic specification is also being developed.

References

[1] A. Aho, J.D. Ullman, R. Sethi. Compilers: Principles, Techniques, and Tools. Addison Wesley. 1986.

[2] S. F. Altschul and B. W. Erickson. Significance of nucleotide sequence alignments. *Mol. Biol. Evol.*, 2:526–538, 1985.

[3] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs, *In JACM 43(1):20-74.*

[4] N. Datta. Semantic basis for Interoperability: An approach based on Horn Logic and Denotational Semantics. MS thesis. NMSU. Aug. 2000.

[5] J. Felsenstein. PHYLIP: Phylogeny inference package, version 3.5c. Distributed by the author, Deparment of Genetics, Univ. Washington, Seattle, 1993.

[6] J.W. Lloyd. Foundations of Logic Programming. Springer Verlag. 2nd ed. 1987.

[7] C. Goldfarb, P. Prescod. The XML Handbook. Prentice Hall. 1998.

[8] `http://www.w3.org/Style/XSL`.

[9] `http://www.w3.org/TR/xslt`.

[10] `http://www.w3.org/TR/voicexml/`

[11] G. Gupta. Horn logic denotations and their applications. In *The Logic Programming Paradigm: The next 25 years*, pages 127–160. Springer Verlag, 1999.

[12] G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium*, San Francisco, pp. 230-239. Dec. 1997.

[13] G. Gupta and E. Pontelli. A Horn logical semantic framework for specification, implementation, and verification of domain specific languages. Essays in honor of Robert Kowalski, Springer Verlag, Lecture Notes in Computer Science, to appear.

[14] G. Gupta, O. El Khatib, M. Noamany. Building the tower of Babel: Converting XML to VoiceXML for Accessibility. Proc. 7th International Conference on Computers Helping People with Special Needs (ICCHP00). OCG Press (Austria). pp. 267-272.

[15] Haifeng Guo. Translating Nemeth Math Braille Code to LaTeX: A Semantics-based Approach. *Master Thesis.* New Mexico State Univ, 1999.

[16] C. Gunter. Programming Language Semantics. MIT Press. 1992.

[17] C. Heitmeyer, et al. Automated Consistency Checking of Requirement Specification. In *ACM Trans. on Software Engg. and Methodology*, 1996.

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

[19] J.R. Iglesias, G. Gupta, E. Pontelli, D. Ranjan, B. Milligan. Interoperability between Bioinformatics Tools: A logic programming approach. In *Proc. Practical Aspects of Declarative Langs*, 2001. Springer Verlag LNCS 1990.

[20] H. Guo, A. Karshmer, G. Gupta, S. Geiger, C. Weaver. A Framework for Translation of Nemeth Braille Code to LaTeX: The MAVIS Project. In *Proc. ACM Conf. on Assistive Technologies*, pp. 136-143, 1998.

[21] L. Liebmann. Extensible Markup Language, XML's Tower Of Babel. `http://www.internetweek.com/indepth01/indepth042401.htm`.

[22] David R. Maddison, David L. Swofford, and Wayne P. Maddison. NEXUS: An extensible file format for systematic information. *Syst. Biol.*, 46(4):590–621, 1997.

[23] Wayne P. Maddison and David R. Maddison. *MacClade: Analysis of phylogeny and character evolution, version 3.07.* Sinauer, Sunderland, Massachusetts, 1997.

[24] K. Miesenberger, B. Stöger. Personal Communication.

[25] A. Nemeth. The Nemeth Braille Code for Mathematics and Science Notation 1972 Revision (Frankfort KY: American Printing House for the Blind, 1972)

[26] R. D. M. Page. COMPONENT, version 2.0. The Natural History Museum, London, 1993.

[27] L. Scadden. Making Mathematics and Science Accessible to Blind Students Through Technology. Proceedings of RESNA'96, 1996.

[28] D. Schmidt. *Denotational Semantics: a Methodology for Language Development.* W.C. Brown Publishers, 1986.

[29] D. Schmidt. Programming language semantics. In CRC Handbook of Computer Science, Allen Tucker, ed., CRC Press, Boca Raton, FL, 1996. Summary version, ACM Computing Surveys 28-1 (1996) 265-267.

[30] L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.

[31] S. Stepney. High Integrity Compilation. Prentice Hall. 1993.

[32] D. L. Swofford. PAUP: Phylogenetic analysis using parsimony version 3.1.1. Illinois Natural History Survey, Champaign, 1993.