Cite as: Ragonis, N., & Shilo. G. (2014). Drawing analogies between logic programming and natural language argumentation texts to scaffold learners' understanding. *Journal of Information Technology Education: Research, 13,* 73-89. Retrieved from http://www.jite.org/documents/Vol13/JITEv13ResearchP073-089Ragonis0415.pdf

Drawing Analogies between Logic Programming and Natural Language Argumentation Texts to Scaffold Learners' Understanding

Noa Ragonis
Beit Berl College and Technion – Israel Institute of Technology, Israel

noarag@beitberl.ac.il

Gila Shilo
Beit Berl College, Israel

gilas@beitberl.ac.il

Abstract

The paper presents a theoretical investigational study of the potential advantages that secondary school learners may gain from learning two different subjects, namely, logic programming within computer science studies and argumentation texts within linguistics studies. The study suggests drawing an analogy between the two subjects since they both require similar abstraction skills manifested in the analysis of texts and in capturing their logic structure and inference. We propose that drawing analogies between two representations of argumentation texts can advance students' understanding, and, furthermore, using computerized systems may enable students to interact with linguistics texts and thus enhance their understanding. The paper explores the connections between the two disciplines, emphasizing the similar structures used to express the knowledge, and presents the similar abstract thinking processes that learners must carry out. Further implications for curricula are discussed.

Keywords: Logic Programming education, Linguistics education, Argumentation texts, Analogies, Abstraction

Introduction

The learning of argumentation texts is included in all educational levels from kindergarten, throughout schools, till academic degrees. Even pre-school children use arguments when trying to justify their claims (Stien & Miller, 1993). Studies show that young students find it difficult to formulate a good argument (Orsolini, 1993). For example, researchers addressed the difficulties of young students aged 9-11 and found that they encounter problems in finding justification for their claims (Berkowitz, Oser, & Althoff, 1987). Studies conducted on older students and adults also found that they experience difficulties presenting eligible justifications and arguing with counterclaims (Kuhn, 1991). Kuhn (1991) also found that students tend to base their claims on explanations more than on evidence. Understanding an argumentation text requires exposure to its structure or, in other words, the ability

Editor: Lynn Jeffrey

to identify the presented argument and to distinguish between the argument and the argument justifications. Argumentation texts can take on different constructions that may make them more difficult to understand. In some structures the inference is concealed in the text, and so it is more complicated for readers to recognize and understand.

Logic programming, a field in computer science, is also based on inference. Inference is based on the way people believe that human inferential thinking is performed, i.e., it is based on basic rules of mathematical logics. For example, if condition *cond1* exists (its logical value is *true*) and condition *cond2* does not exist (its logical value is *false*), then the following composite conditions can be inferred: (*cond1 and cond2*) is *false*, (*cond1 or cond2*) is *true*, (*not cond2*) is *true*. This nature of inference can be observed in the framework of logic programming, both in the inference engines of the logic programming languages and in the way programs are written as knowledge bases, described by facts and logic rules.

We claim that the abstract abilities required to understand argumentation texts are similar to those required to formalize problems in logic programming language. Since logic programming is based on elementary structures that capture the formation of argumentation texts, mastering it may enable students to advance their understanding of such texts. Such enhanced understanding may be gained based on learning from different representations of the texts and from the opportunity students have to use computerized systems that enable them to interact with the texts. Students can also formalize their interpretation of the inference presented in an argumentation text as a logic program. Furthermore, by using logic programming as a knowledge base, students can present queries and get answers that validate or disprove their assumptions, and thus enable them to check their preceding interpretation.

The broad intention of our study is to investigate the possibility of scaffolding students' understanding of texts and knowledge that involves inferring comprehension by presenting explicit analogies between the two disciplines. We believe that providing learners with tools from the two different disciplines, and displaying the similar connections between the knowledge entities as presented in both disciplines, can improve learners' understanding and ability to cope with knowledge that involves inference.

In this paper we conduct a theoretical investigation comparing the structures of a natural language argumentation text with the structure of a program written in logic programming language (Prolog). Although the study is based on comprehensive experience and research involving learning processes of the Hebrew language, we believe our conclusions characterize any natural language since the study refers to logic thinking processes that are based on discovering the abstract structure of texts. The logic programming Prolog has several different versions based on natural languages such as Russian, Japanese, and Hebrew. This reinforces the assertion that the discussion is language independent. In our research we focus on texts that are used regularly in the teaching of natural languages and show the potential gain to students from transferring the acquired knowledge between learning logic programming within the framework of computer science studies and argumentation texts within the discipline of language studies.

Literature Review

Linguistics and Computer Science

The two disciplines of linguistics and computer science have many converging points. Regarding the computer as an implementation tool and as a research tool in linguistics revealed the following three main mutual influences: (1) Theories and research methodologies that are grounded in computer science have been adopted by linguistics research, for example, the description of formal languages using automata, a model used in computer science; (2) Theoretical assertions about

the similarities between computer computational methods and the way humans learn and use natural language, have led to the use of various computer science tools, specifically artificial intelligence tools. An example of such theoretical work is Rahwan and Simari's book entitled "Argumentation in Artificial Intelligence" (2009), which presents different ideas and specifically relates to the subject of argumentation; and (3) Computerized systems that simulate human linguistic behavior, such as translation or the understanding of natural language questions that can be used with any search system, have been developed and are referred to as natural language processing. An example is research which concerns fuzzy logic and offers computational tools that conclude in non-objective or fuzzy conditions or that allocate a suitable sense to sentences that include fuzzy prepositions like "few" or "often" (Freksa, 1994; Zadeh, 1997).

In 1957, Chomsky introduced the theory of formal languages and expanded it in later works (Chomsky, 1965, 1975). Chomsky theory enables the definition of a set of rules that facilitates the building of all valid sentences in the language. This theory, which came from the linguistics research on natural languages, came to be a central tool in mathematics theory and in computer science theory with respect to programming languages. Research that combines the two disciplines was established and called 'computational linguistics,' and academic institutes began to offer formal degrees in this discipline. Computational linguistics deals with the development of computerized tools to cope with natural language. The theoretical basis for these developments rests on the attempts to identify formal structures in natural languages and to further investigate how such structures serve as a basis for computational inference as executed by a computer program. The objective is to find algorithms that address issues involving natural languages, such as automated translation between different natural languages, computer dialogs in natural language, analysis of documents, and the understanding of natural language.

In the 1950s and 1960s, in parallel with the development of the formal languages theories another field, artificial intelligence (AI), also developed (Bratko, 1990; McCarthy, 1958; Sterling & Shapiro, 1994) and influenced the field of computational linguistics. AI research focused on logics and deductions, which led to the identification of models that capture natural language semantics and to the development of the first computer implementations that captured the meaning of natural language, such as Prolog (PROgramming in LOGic). Up until that time, the vast majority of computer programs were written using formal computer languages rather than natural languages. Prolog is exceptional in its use of natural language words as part of the program code and in its syntax, which is very similar to that of natural language conditional sentences. The language operations are very minimalist and are based on human logical inference (if <conditions> - then <operations>), hence the language is referred to as a logic programming language. Prolog's suitability for analyzing natural language structures led to the development of various CALL (computer-assisted language learning) software systems. One project, for example, used C-Prolog to evaluate the correctness of simple English sentences that were based on some of the main Prolog characteristics, such as being a non-numeric programming language, offering the advantages of user-friendliness and being, above all, descriptive (relational) rather than procedural (Butcher, Galletly, & Wong, 1990). Dung (1995) asserted that logic programming is an ideal environment for implementing data bases and presented an in-depth exploration of logic formalization as a computational mechanism that enables investigation of human natural inference when constructing and understanding an argument. Prolog was used in the development of an intelligent computer-assisted language learning (ICALL) system for learning Arabic, designed to be used by students at primary schools or by learners of Arabic as a second or foreign language (Shaalan, 2005). Prolog was chosen as the implementation language since the language's grammatical rules can be specified using Horn clauses and because the Prolog interpreter uses the strategy of depth-first top-down parsing algorithm, which fits language structures. This strategy is best also for presenting argumentation texts, as in our analyses.

Based on the above interpretations, we further present correspondence between the structure of argumentation texts taught in linguistic educational settings and the structure that enables presentation of argumentation texts in Prolog, in computer science educational settings, and offer analogies between the two disciplines to enhance learners' understandings of such texts. Our interpretation is different from the uses of CALL systems. We claim that using the Prolog language as-is, is most suitable for the interpretation of argumentation texts. We wish to take advantage of the fact that Prolog is accessible and easy to learn, and in some countries is taught within the CS curriculum.

Logic Programming in Education

Computational thinking is nowadays appreciated, as a tool for thinking and inferring, in many other disciplines other than computer science, and many countries strive to integrate it into curricula at all levels (e.g., Wing, 2006). Scientists cope with the challenge and need to integrated disciplines and, particularly, to meet the need to adopt computational representations and algorithms used in computer science to other domains. A reflection of this can be seen, for example, in a multinational research work conducted by leading scientists from different countries (Microsoft Research, 2006) and in a collection of research papers that includes the two said disciplines (Martín-Vide & Mitrana, 2001).

Work has been done in educational frameworks on knowledge representation and logic representation in logic programming. One heterogeneous study, for example, presented both theoretical and empirically based findings using the framework of logic programming (Habiballa & Kmet', 2008). This study highlighted the key role of logic in computer science, computer science education, and knowledge representation but failed to take the discussion any further, i.e., to an integration with what is considered to be a different discipline – linguistics. Another study presented a methodology for teaching logic programming using analogies (Lopez, 2001). This study suggested giving students declarative programming examples that illustrate various concepts and then asking them to write their own programs using an analogy process. The analogies used in this study were based on similar relations within different contexts rather than on two different, though similar, representations in two different disciplines.

The analogy we suggest implementing in educational settings is based on the fact that logic programming using Prolog is already used in middle and high schools worldwide (Bottino, Forcheri & Molfino, 1995; Cope, 1989). Another issue dealt with by researchers is how to cope with the pedagogy of teaching logic programming (Di Bitonto, Roselli and Rossano, 2009; Stamatis & Kefalas, 2007). Linck and Schubert (2011), for example, investigated the new German curriculum for teaching logic programming in secondary education. These researchers' aim was to improve informatics in secondary schools based on a model of logic programming competence levels. In Israel, a broad high school logic programming curriculum is already established and includes various advanced subjects such as artificial intelligence and expert systems (Haberman, Shapiro & Scherz, 2002; Haberman & Scherz, 2005; Ragonis, 1996; Ragonis, Scherz, Ben-Ari, & Shapiro, 1998; Scherz, Haberman, Ragonis, & Shapiro, 1993; Scherz, Haberman, & Ragonis, 1994).

Disciplinary Background

Argumentative Texts

The meaning of argumentative texts

An argumentative text is a text in which the addresser presents a claim and is then required to prove it in order to convince the addressee of the validity of his or her claim. In the text, the ad-

dresser makes an assumption that leads to a conclusion and supports a particular opinion using different methods of justification such as explanations, examples, and comparisons (Antaki, 1994; Brooks & Warren, 1972; Copi, 1982; van Dijk, 1980). Argumentative texts differ from expository texts, which convey information about events, facts and ideas, interpret historical events, and clarify opinions without presenting the writer's own opinion (Goelman, 1982; Sarel, 1991, Shilo, 2003).

Several researchers described the structure of argumentative texts: Toulmin (1969) presented a five-part model that included the possibility of a counter-argument, which was further investigated in modern Hebrew (Alon, Grilac, & Shilo, 2006; Livnat, 2011). Mann and Thompson (1998) introduced the Rhetorical Structure Theory (RST) that provided a method of describing the relations between clauses in a text according to whether they are grammatically or lexically signaled. The RST is a useful framework for relating the meaning of conjunctions, the grammar of clause combining, and non-signaled parataxis. Mann and Thompson (1998) revealed semantic relations that repeat themselves throughout the various texts and demonstrated how RST can be used to identify the main idea of a text. Azar (1999) described the argumentative text in the context of RST and claimed that the nucleus-satellite relation is the most important structure of such texts. According to Azar (1999), the nucleus is the writer's main purpose and the satellite adds details that supplement the nucleus in various ways, for example, by convincing. Azar refers to five subtypes of argumentative satellites: evidence, justification, motivation, antithesis, and concession. In his opinion, the purpose of the first three subtypes is to convince, while the latter two present a claim and then negate or refute it.

Learning argumentation was viewed by researchers not only as a one-off activity but as a stage-wise and collaborative process that builds on the students' concrete experience, reflective thinking, and observation over a period of time (Ho, Mei Lin, Natasha, & Chee, 2009). Ho et al. (2009) used the Second Life immersive virtual environment as a platform, and, following their work, we recommend using Prolog programming language as the environment on which to build the stage-wise reflective learning activity.

In a previous paper (Shilo & Ragonis, 2014), we presented argumentative texts formatted in a basic claim structure. In such texts, the claim usually appears in the first part of the text but can also appear at the end. In this paper, we present a structure that is based on a basic claim structure (claim and justification) but also contains a claim that opposes the view of the addressee, i.e., a counter structure. In other words, the addressee presents a claim, the opposing argument, and the justification of the claim. This structure is considered more influential and convincing than a basic claim because the addresser presents his or her claim explicitly while adding a counter-claim. Since the addresser is familiar with the subject, he or she can then proceed to refute the counter-claim.

Basic structures of argumentative texts

Argumentative structures have one of three common basic constructions: basic claim structure, counter structure, and "pros and cons" structure. We will first present the three structures and then elaborate on the counter structure, whose representation, in both natural language and logic programming language, will be interpreted in detail.

A. Basic claim structure: The aim of the basic claim structure is to convince the addressee that the addresser's claim is valid. The claim is the main idea of the text, whose structure is usually introduction, claim, justification, and summary, which reiterates the claim. A less common way of presenting the claim involves presenting it at the end of the text. In this case, the order of the text's parts will be introduction, justification, which leads to the claim at end, which in turn serves also as a summary.

B. Counter structure: The counter structure presents both a claim and a counter-claim according to the following order: introduction, counter-argument, the claim and justifications for the claim. Sometimes the addresser's claim is presented before the counter-argument and may even include explanations of the counter-argument. In any case, the counter-argument must always appear before the justification of the writer's claim, as will be elaborated on later.

C. "Pros and cons" structure: This structure, in which two opposing arguments are presented, is used mainly in debates and discussions. The paragraph opens with an introduction, in which the subject and a clue and/or an explicit statement about the dispute are presented. This is followed by the presentation of one claim after which the counter claim is presented, followed by a conclusion of one sort or another (agreement with one of the claims, a new viewpoint, a compromise, or a standoff).

The counter structure

The counter structure seems to be more convincing than the basic claim structure since a text that contains an argument opposing the writer's claim is more persuasive than an ordinary text containing only a difference of opinion. The presentation of the counter-argument indicates that the addresser is confident, has explored all possibilities, has reached his or her conclusion, and is not afraid to address the opponents' claims or even confront them. The opposing argument generally appears in the opening section, before or after the addresser's claim, but never after the supporting argument, so as not to interfere with the persuasion process. After the supporting argument is presented, no new arguments are introduced and the addresser concludes with a recap of the original argument presented at the beginning of the argument, thereby completing a cycle.

The five parts of the texts in the counter structure are:

1. Introduction
The introduction can be either a presentation of the topic and/or of a problem, background information such as a summary of theories, or an example or a story meant to attract the reader as a "teaser".

2. Addresser's claim

3. Counter-claim

4. Justification of the addresser's claim
The justification of the addresser's claim can consist of details, exemplifications, data, grounds and/or definitions.

5. End
The end paragraph of counter structure can be a summary, a conclusion, a recommendation, a prediction or any combination of them.

Logic Programming

What is logic programming?

A computer program is an implementation of an algorithm that is written in a programming language and developed in order to solve a problem. A programming language consists of data structures and control structures that enable manipulation of the data. Programming languages are attributed to programming paradigms that differ in their principles of knowledge representation and in their ways of execution (Detienne, 2001). The differences between the paradigms are first reflected in the way a given problem is analyzed. The logic programming paradigm is essentially different from other paradigms (e.g., procedural, functional, object oriented) since it uses struc-

tures and inferences that are commonly used in mathematical logics and are accepted as being similar to human logical thinking. The logic programming paradigm is based on first-order predicate calculus. This programming style emphasizes the declarative description of a problem rather than the decomposition of the problem into an algorithmic implementation. A logic program is a collection of logical declarations that describe the problem to be solved, whereby the problem description is then used by an inference engine to find a solution. Logic programming is restricted to backwards chaining in the form usually referred to as a *rule*. Rules represent logical connections between claims and are presented using logic syntax. The main structure of a rule is as follows: *G if G1 and G2... and Gn*, where *Gi* is a goal. This structure is behaves like a goal-reduction procedure, and its semantics are *to solve G, you have to solve G1 and ... and Gn*. The goal *G* is called the *rule head*, and the combined logic claim *G1 and G2 ... and Gn*, is called the *rule body*. In other words, if the rule body is valid, then the head of the rule is valid. Every goal *Gi* can either be derived from another rule or can be defined as a *fact*, which means that it exists, its logic value is true, and so it needs no further reduction. A *fact* can also represent relations between values. Thus, a logic program is a set of facts and rules that can be derived based on each other, combined with logical operators such as *and*, *or*, and *not*. Computation in logic programming is in fact a proof search, which determines whether proof can be derived for a given goal. The given goal is presented to the program as a *query* and the built-in *inference engine* of the language searches a proof to solve the query. If a proof (a chain or tree of goals) can be derived based on the specific program and the proof is deemed successful then the answer to the query is *yes*; otherwise, if the proof fails, the answer to the query is *no*. Knowledge in logic programming is presented in terms of facts and rules, referred to as a knowledge base (which is actually a program). The language, based on its built-in inference engine, can follow inferences and determine either that a goal can be derived from the knowledge base, i.e., the deduction is valid, or that it cannot be derived, i.e., the deductive is invalid. The most commonly used logic programming language is Prolog (PROgramming in LOGic). This declarative language is based on first-order logic, which is used in artificial intelligence applications. The common concept known as "running a program" does not exist in logic programming. Rather, a query is presented and the technical mechanisms *unification* and *backtracking* together serve the inference engine, which outputs an answer whether the query can or cannot be derived from the knowledge base. A simple and traditional example of logic program that relates to family relations and demonstrates how natural language serves the programming language is presented in the Appendix. The example shows how a Prolog program is similar to a text written in a natural language and how the inference executed is similar to basic human logical thinking inference.

Steps in developing a logic program

When representing information as a logic knowledge base – a logic program - the next phases are carrying out the following:

Step 1 – *Definition of targets:* The subject and the objectives of the program are defined, or in other words, what are the main objectives that will be further presented as queries, on which we want to get answers in relation to the knowledge base. All of the other knowledge structures will be defined based on those definitions.

Step 2 – *Choosing descriptors:* Since the inference is based on facts, the fact structures must first be defined. The rule definitions will rely on the fact structures. Specific data that is written in the program is not, in itself, of importance, but the structure that represent the relations between the given data components is essential.

Step 3 – *Choosing the relations:* The relations – the rules heads – are now defined based on the objectives defined in Step 1. In this stage, a logic relation is expressed for each of the rule heads.

The objective of the goal, and the relations (rules or facts) on which it is based, are of importance here.

Step 4 – *Formalization in Prolog, programming:* In this step the Prolog program is written – facts and rules. The specific data is presented using both the fact structures chosen in Step 2 and the rule heads and their logic declarations chosen in Step 3.

Step 5 – *Demonstrating queries (running the program):* After implementing all facts and rules, queries relating to the different objectives may be presented. The answers given by the inference engine can be either yes or no, according to the specific inference validation. If the logic answer is yes – some specific results may be obtained as well.

Demonstrating the Analysis of Argumentation Structure in Linguistics and in Logic Programming

In this section we will use an example of a counter-structure argumentation text to demonstrate text analysis according to the text's linguistics structure and its representation in logic program (Figure 1).

Recently the subject of conducting experiments in animals has surged again. Some people object to animal experiments and claim that they reflect abuse and that animals should be treated like humans. Others believe that these experiments are necessary in order to promote basic research, add to our understanding of physiological and pathological processes, and are crucial for the development of new drugs and therapies. Despite the claims of the opponents, no substitute has been found to conducting experiments in animals, hence research experiments must continue.

Figure 1: Example of a counter structured text

Analysis of Counter Structure in Linguistics

Table 1 presents the analysis of the text presented in Figure 1 according to the counter structure presented in the section entitled Disciplinary Background – Basic Structures of Argumentative Texts.

Table 1: Expressing the example text in counter structure form

Counter structure part	Quote from the example text
1. Introduction	*"Recently the subject of conducting experiments in animals surged again"*
2. Addresser's claim	*"Others believe that these experiments are necessary"*
3. Counter-claim	*"Some people object to animal experiments and claim that they reflect abuse and that animals should be treated like humans."*
4. Justification of the addresser's claim	*"in order to promote basic research, add to our understanding of physiological and pathological processes, and are crucial for the development of new drugs and therapies"*
5. End	*"Despite the claims of the opponents, no substitute has been found to conducting experiments in animals, hence research experiments must continue."*

Expressing the Counter Structure in Logic Programming

In this section, the text presented in the example is formalized in a Prolog program. We first present the program elements, and then display the full Prolog program and explain the way the inference is carried out.

The Prolog program

As presented in the section Disciplinary Background – Steps in Developing a Logic Program, the first step in analyzing a text in order to present it as a Prolog program is to determine the objective of the inference. In the above example text, the subject is "experiments in animals" and the objective is to conclude whether or not it is reasonable to conduct experiments in animals. Next, it is necessary to distinguish between elements that are already known to be true, which are to be presented as facts, and elements that must be inferred, and so are presented as rules. The text contains justifications for two different claims, those that support the conducting of experiments in animals and those that do not. In the counter structure presented here in detail, only justifications for the addresser's claim are presented.

To enable fluent inference and generalization, the first fact formation to use is claim. Claims can express either agreement or disagreement. This knowledge is presented in the following clauses:

claim(agree).

claim(disagree).

The main elements presented in the text are the different justifications. To present each justification we use the relation *standpoint*. This relation has two descriptors, a claim and a related justification. For example, the supportive text *"experiments are necessary in order to promote basic research,... "* is presented by the following clause :

standpoint(agree, experiments_are_necessary(to_promote_basic_research)).

Syntax remarks: (a) the underline between the words is necessary in this programming language to indicate that all of the words are a single element; (b) use of brackets (…) enables further inference that is not presented here.

And the non-supportive text *"Some people object to animal experiments and claim that they reflect abuse,... "* is presented by the following clause:

standpoint(disagree, abuse).

All of the other justifications are presented similarly, each as a separate fact.

The objective of the program is to infer from the facts and to draw a conclusion. In the chosen formalization we will use three main rules:

1) A rule that collects all justifications for a specific claim, named justifications. A list of justifications that supports the claim is, therefore calculated for each claim. The head of the rule is: *justifications(Claim, Justifications_list)*.
2) A rule that calculates the number of justifications addressed for each claim, named *justifications_count*. The number of justifications that support the claim is calculated for each claim. The head of the rule is: *justification_counts(Claim, Number)*. The rule inference is based on the previous rule (Rule 1), which calculates the list of justifications, and further calculates the number of elements within this list.
3) The main rule that drives the conclusion about the text argument is: *conducting_experiments_in_animals*. The conclusion in this demonstration is based on a simple decision about whether the number of supportive justifications is greater than the number

of non-supportive justifications. The rule inference is based on the previous rule (Rule 2), and calculates the number of justifications for each claim.

Figure 2 displays the program at a glance.

```
% claim(Claim: agree / disagree).
claim(agree).
claim(disagree).

% standpoint(Claim, Justification).
standpoint(agree, experiments_are_necessary(to_promote_basic_research)).
standpoint(agree, experiments_are_necessary(to_understand_physiological_processes)).
standpoint(agree, experiments_are_necessary(to_understand_pathological_processes)).
standpoint(agree, experiments_are_crucial(for_the_development_of_new_drugs)).
standpoint(agree, experiments_are_crucial(for_the_development_of_new_therapies)).
standpoint(disagree, abuse).
standpoint(disagree, animals_should_be_treated_like_humans).

justifications(Claim, Justifications_list):-
                claim(Claim),
                findall(Justification, standpoint(Claim, Justification), Justifications_list).

justifications_count(Claim, Count):-
                justifications(Claim, Justifications_list),
                list_count(Count, Justifications_list).

conducting_experiments_in_animals:-
                justifications_count(agree, Count1),
                justifications_count(disagree, Count2),
                Count1 > Count2.
```

Syntax remarks: (1) the notation % indicates that the line is a remark line and serves to document the structure ahead; (2) an element that starts with a lower-case letter is a constant; an element that starts with a capital letter is a variable. For example, the variable Claim can have the values *agree* or *disagree*, and the variable Count can be any number that is calculated.

Figure 2: The Prolog program formalizing the counter-structure text example

Examples of the program outputs

The developed program enables to present different kinds of queries. We will limit our presentation to three main rules. Figure 3 presents the queries, their answers, and short explanations.

Query No. 1: For each claim (agree or disagree), give the list of justifications

?- justifications(Claim, Justifications_list).

Claim = agree
Justifications_list = [experiments_are_necessary(to_promote_basic_research) ,
experiments_are_necessary(to_understand_physioloical_processes) ,
experiments_are_necessary(to_understand_pathological_processes) ,
experiments_are_crucial(for_the_development_f_new_drugs) ,
experiments_are_crucial(for_the_development_of_new_therapies)] ;

Claim = disagree
Justifications_list = [abuse , animals_should_be_treated_like_humans]

Query No. 2: For each claim (agree or disagree), give the number of justifications (i.e., the size of the list produced in Query No. 1)

?- justifications_count(Claim, Count).

Claim = agree
Count = 5 ;

Claim = disagree
Count = 2 ;

Query No. 3: Give a final conclusion whether or not animal experiments should be conducted (this is based on the number of justifications for each of the opposing claims).

?- conducting_experiments_in_animals.
Yes

Figure 3 : Queries and their answers

Explanation of the inference process

Each of the three rules is logically based on previous rules or on the facts. When a query such as *conducting_experiments_in_animals*, is presented, the language inference engine identifies this constant as a head of rule, and so in order to reach a conclusion, the rule body must be valid. This rule body determines the number of *agree* justifications and the number of *disagree* justifications, and calculates whether there are more *agree* justifications than *disagree* justification. In order to determine the number of *agree* and *disagree* justifications, a suitable rule, *justifications_count*, is in place that gives the required answer. The *justifications_count* rule uses the *justifications* rule that provides a list of justifications for each specific claim and uses a built-in descriptor that calculates the size of each such list. Thus, the *justifications* rule uses a built-in descriptor that identifies all of the facts with the structure, *standpoint(Claim, Justification)*, and collects them into a list. So, a tree of inference using the language mechanism is constructed in order to answer our main objective question – whether or not to conduct experiments on animals, to which the final answer is yes.

Reflection of the Prolog program in the linguistics counter structure

As argued earlier, there is a clear analogy between the counter structure and its formalization in Prolog. Table 2 presents this analogy.

Table 2: Expressing the logic program in the counter structure form

Counter structure part	Quote from the program
1. Introduction	Does not appear in the program but is the program's title – what the program is about.
2. Addresser's claim	Facts with the structure: *standpoint(**agree**, << justification >>).*
3. Counter-claim	Facts with the structure: *standpoint(**disagree**, << justification >>).*
4. Justification of the addresser's claim	Rules that build on each other: *justifications(Claim, Justifications_list):- ...* *justifications_count(Claim, Count):- ...* *conducting_experiments_in_animals:- ...*
5. End	The target query and its answer : *?- conducting_experiments_in_animals.* *yes*

Comparison between the Representations

A comparison between the two representations in Tables 1 and 2 reveals several immediate similarities. It can be seen that the addresser's claim and the counter-claim are "known" from the text and since they appear as facts in the Prolog program. The justification of the addressor's claim is reflected in the Prolog program by the definition of rules, which are the heart of the required logic inference, and are actually independent of the textual content. The rules formalize what is actually happening in the human mind upon reading a counter text and generating conclusions. The opportunity to display a query in the Prolog environment enables learners to check whether or not their own conclusion is valid. In fact, the display of the query corresponds fully to the "end" part of the argumentation text. In the counter-structure text, the addresser's claim is reiterated at the end in order to finalize the argument. In the Prolog environment, a query must to be presented in order to receive an answer. The query is a trigger for a sequence of inferences based on the rules which ends with the facts.

As can be seen, the structure of the text that must be uncovered in both disciplines is similar. Students must use abstract abilities when reading the text in order to cope with its meaning, the argument. They must discover what the claim is, what justifications are displayed, and what can be concluded (inferred) based on that. The process is essentially the same in both representations, though the way of formalizing the text is different. Thus, by alternating between the two forms of representation and by highlighting the analogies, the two different representations of the text may serve to mutually develop students' skills and so enhance their understanding of texts.

Summary

The paper suggests a theoretical correspondence between two ways of representing argumentation texts. The process, we believe, can support teachers by presenting analogies between two different, though similar, representations of argumentation texts in language studies and in computer science studies, particularly logic programming. Teachers in each of the two disciplines can use such analogies without being too concerned about their veracity since, as teachers of one discipline, they are not actually required to master the other discipline. The teaching-learning processes can rely on the students' knowledge, and teachers can lead discussions and use them to

formally demonstrate analogies and, thus, develop the students' skills. We believe that it makes no difference which discipline is learned first; in either case connections can be made and pointed out. The analogies demonstrated in the paper can be further applied to the discipline of mathematics and, specifically, to geometric proofs, which are studied in high school but are considered to be relatively difficult for students to understand. In geometry, for instance, the frequently used phrase "need to prove" is the claim; the partial relations between mathematical elements referred to as "the proof" are actually the justifications; and the phrase used to finalize the proof, i.e., "what was needed to be proved", is the end of the argument that must also relate to the claim.

We argue that learning argumentation texts by exposing and emphasizing their structure, alternatively in the two disciplines of computer science and linguistics, will develop the understanding of the semantics of those texts and further develop the mathematical logical thinking about which we wish to expand our investigation.

Based on the theoretical investigation presented in the paper, we intend to further examine this topic by performing field research. The research population will be middle or high school students, and the objective will be to investigate the mutual understanding of students who study both disciplines while the appropriate analogies are presented and discussed in class. The research will examine students' ability to understand and use such analogies and will determine whether or not the first discipline studied has an influence on these abilities. Such research may also examine the teachers' ability to cope with (slightly) diverse challenges in their classrooms. Since, we expect teachers to deliver the analogies between the two different subjects to their students, it is important to investigate their positions regarding this new approach.

References

Alon, I. Grilac, L., & Shilo, G. (2006). *The written text in modern Hebrew*. Tel Aviv, Israel. Thema-Mofet Press (in Hebrew).

Antaki, C. (1994). *Explaining and arguing*. London: Sage.

Azar, M. (1999). Argumentative structures. In A. Ornan, R. Ben-Shar, & G. Tori (Eds.),. *Hebrew as a living language.* Tel Aviv: Hakibutz Hameuhad Press (in Hebrew).

Berkowitz, M. W., Oser, F., & Althoff, W. (1987). The development of sociomoral discourse. In W. Kurtines & J. Gewirtz (Eds.), *Moral development through social interaction*. New York: John Wiley & Sons.

Bottino, R. M., Forcheri, P., & Molfino, M. T. (1995). Logic programming and education. *The Knowledge Engineering Review*, *10*, 209-211.

Bratko, I. (1990). *PROLOG Programming for artificial intelligence* (2nd ed.). Wokingham, England: Addison Wesley.

Brooks, C., & Warren, P. W. (1972). *Modern rhetoric*. San Francisco: Harcourt Brace and World.

Butcher, W., Galletly, J., & Wong, A. (1990). Prolog and language analysis: Intelligent response to comprehension replies. *Computer Assisted Language Learning*, *3*(1), 27-36.

Chomsky, N. (1957). *Syntactic structures.* The Hague/Paris: Mouton.

Chomsky, N. (1965). *Aspects of the theory of syntax*. Cambridge, Massachusetts: MIT Press.

Chomsky, N. (1975). *"Introduction", The logical structure of linguistic theory.* New York: Plenum Press.

Cope, P. (1989). Teaching PROLOG programming in the secondary school. *Journal of Computer Assisted Learning*, *5*, 223–230.

Copi, I. M. (1982). *Introduction to logic*. Tel Aviv, Israel: Ychdav Press.

Détienne, F. (2001). *Software design – Cognitive aspects*. (F. Bott translator and editor). Heidelberg, UK: Springer.

Di Bitonto, P., Roselli, T., & Rossano, V. (2009). Formative evaluation of a didactic software for acquiring problem solving abilities using Prolog. In P. Paolini & F. Garzotto (Eds.), *Proceedings of the 8th International Conference on Interaction Design and Children*, ACM, New York, 154-157.

Dung, P. M. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, *77*(2), 321-357.

Freksa, C. (1994). Fuzzy logic: An interface between logic and human reasoning. *IEEE Expert*, 9(4), 20-21.

Goelman, H. (1982). Selective attention in language comprehension. *Discourse Processes*, *5*, 53-72.

Haberman, B., & Scherz, Z. (2005). Evolving boxes as flexible tools for teaching high-school students declarative and procedural aspects of logic programming. *Lecture Notes in Computer Science*, *3422*, 156-165.

Haberman, B., Shapiro, E., & Scherz, Z. (2002). Are black-boxes transparent? High school students' strategies of using abstract data types. *Journal of Educational Computing Research*, *27*(4), 411-436.

Habiballa, H. and Kmet', T. (2008). Mathematical logic and deduction in computer science education. *Informatics in Education*, *7*(1), 75-90.

Ho, C.; Mei Lin, R., Natasha, A. & Chee, Y. S. (2009). Designing and implementing virtual enactive role-play and structured argumentation: promises and pitfalls. *Computer Assisted Language Learning*, *22*(5), 381- 408.

Kuhn, D. (1991). *The skills of argument*. New York: Cambridge University Press.

Linck, B., & Schubert, S. (2011). Logic programming in informatics secondary education. *Proceedings of 5th international conference ISEEP 2011*, 24, (12p. on CD).

Livnat, Z. (2011). *Rhetoric of the scientific article*. Ramat-Gan: Bar-Ilan University Press. (in Hebrew).

Lopez, A. M. (2001). Supporting declarative programming through analogy. *Journal of Computing Sciences in Colleges*, *16*(4), 53-65.

Mann, W. C., & Thompson, S. A. (1998). Rhetorical structure theory: Toward a functional theory of text organization. *Text*, *8*(3), 243-283.

Martín-Vide, C., & Mitrana, V. (Eds.) (2001). *Where mathematics, computer science, linguistics and biology meet*. Boston: Kluwer Academic Publishers.

McCarthy, J. (1958). Programs with common sense. *Symposium on Mechanization of Thought Processes*. National Physical Laboratory. Teddington, England.

Microsoft Research. (2006). *Towards 2020 sciences*. Retrieved 12/2/2012, from: http://research.microsoft.com/en-us/um/cambridge/projects/towards2020science/downloads/T2020S_ReportA4.pdf

Orsolini, M. (1993). Dwarfs do not shoot: An analysis of children's justifications. *Cognition and Instruction*, *11*, 265-280.

Ragonis, N. (1996). *Introduction to expert systems*. Weizmann institute of science and The Israeli Ministry of Education (In Hebrew).

Ragonis, N., Scherz, Z., Ben-Ari, M., & Shapiro, E. (1998). Development, implementation and evaluation of a course in expert systems for high-school students. *ACM SIGCSE Bulletin*, *30*(3), 300 (poster).

Rahwan, I., & Simari, G. R. (Eds.). (2009). *Argumentation in artificial intelligence*. Dordrecht Heidelberg London New York Springer.

Sarel, Z. (1991). *An introduction to discourse analysis*. Israel: Or-am Press (in Hebrew).

Scherz, Z., Haberman, B. & Ragonis, N. (1994). Introduction to logic programming: The development of a multilevel curriculum. P*roceedings of the 7th ICLP workshop on Logic Programming in Education*, Santa-Margarita, Italy, June 1994.

Scherz, Z., Haberman, B., Ragonis, N. & Shapiro, E. (1993). Expert Systems by High School Students in PROLOG Environment. *Proceedings of the 7th International PEG Conference*, Edinburgh, Scotland, July 1993.

Shaalan, K. F. (2005). An intelligent computer assisted language learning system for Arabic learners. *Computer Assisted Language Learning*, 18(1-2), 81-109.

Shilo, G. (2003). *Comprehension and writing of the text*. Reches Even Yehuda, Israel (in Hebrew).

Shilo, G., & Ragonis, N. (2014). Develop learners understanding of natural language argumentation texts by acquiring logic programming skills. *Dapim 57*, Tel Aviv, Israel. Mofet Press (in Hebrew) (In press).

Stamatis, D., & Kefalas, P. (2007). Logic programming didactics. *Proceedings of the Informatics Education Europe II Conference*, 136-144.

Stein, N. L., & Miller, C. A. (1993). The development of memory and reasoning skill in argumentative contexts: Evaluating, explaining, and generating evidence. In R. Glaser (Ed.), *Advances in instructional psychology*, *4*, 285-335. Hillsdale, NJ: Lawrence Erlbaum

Sterling, L., & Shapiro, E. (1994). *The art of Prolog* (2nd ed.). Cambridge, MA: MIT Press.

Toulmin, S. (1969). *The uses of argumentation*. Cambridge, England: Cambridge University Press.

van-Dijk, T. A. (1980). *Macrostructures: An interdisciplinary study of global structures in discourse*. Hillsdalle, NJ: Erlbaum.

Wing, J. M. (2006). Computational thinking. *Communication of the ACM*, *49*(3), 33-35.

Zadeh, L. (1997). Toward a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic. *Fuzzy Sets and Systems*, *90*(2), 111-127.

Appendix
An Example of a Logic Program – Family Relations

One of the traditional examples of the use of natural language in logic programming is the use of a logic program to represent the relations between family members. When relating to family members we usually distinguish between males and females, so the two basic fact structures will be *male* and *female*. The fundamental relation in family is parenthood, so one more fact structure will present this relation using the descriptor *parent*. Next we can express ever more specific relations in a family, such as *father* – a male family member who is a parent; *mother* – a female family member who is a parent; *grandfather* – a male who has a descendant who is a parent, etc. These relations are be defined as rules and it is evident that all relations defined as facts and rules are properties of any family. In this example we will use the original biblical family, but it should be apparent that in order to infer about a different family, only the values presented in the facts need be changed. All of the structures, facts, descriptors and rules do not change. (Some basic syntax remarks are presented in the footnote.)

```
% facts
male(abraham).                          meaning: abraham is a male
female(sarah).                          meaning: sarah is a female
male(isaac).
male(jacob).
parent(abraham, isaac).
parent(sarah, isaac).
parent(isaac, jacob).

% rules
father(X, Y) :- male(X), parent(X, Y).  meaning: if X is male and X is the parent
                                                 of  Y, then X is the father of Y
mother(X, Y) :- female(X), parent(X, Y).
grandfather(X, Y) :- father(X, Z), parent(Z, Y).

% query examples
?- mother(X, Y).
X = sarah
Y = isaac

?- grandfather(X, Y).
X = abraham
Y = jacob
```

Syntax remarks: In Prolog: (a) a constant must start with non-capital letter (e.g. abraham); (b) a variable that serves to identify relations within rules must start with capital letter (e.g. X); (c) the symbol :- meaning “if”; (d) the symbol ; meaning “and”; (e) a dot indicates the end of any claim – fact or rule.

Biographies

Dr. Noa Ragonis is head of the Instructional Development Center at Beit Berl College. She is a senior lecturer at the Department of Computer Science, Beit Berl Academic College, and at the Department of Education in Technology and Science, Technion. Dr. Ragonis is active in educational research, focusing on cognitive aspects of teaching and learning of Computer Science, and is also involved in pre-service and in-service teachers preparation programs. She is co-author of the *Guide to Teaching Computer Science* (2011, Springer) and has authored eight Computer Science high-school text books and teachers guides.

Dr. Gila Shilo is head of the Division of Cultural Studies and Humanities at the Faculty of Society and Culture, Beit Berl Academic College. In addition she is senior lecturer at the Department of Hebrew Language and at MOFET Institute School of Professional Practice. Dr. Shilo is active in language and educational research and published numerous articles and four books. She has organized and actively participated in numerous national and international meetings.