

TRIANGLE

LLENGUATGE, LITERATURA, COMPUTACIÓ
LANGUAGE, LITERATURE, COMPUTATION

1

Introduction to Language and Computation

Leonor Becerra-Bonache
Henning Christiansen
Veronica Dahl

TRIANGLE 1

September 2010

Introduction to Language and Computation

Leonor Becerra-Bonache
Henning Christiansen
Veronica Dahl

LLENGUATGE, LITERATURA, COMPUTACIÓ
LANGUAGE, LITERATURE, COMPUTATION



Tarragona, 2010

Revista TRIANGLE

President: Antonio Garcia Español

Consell editorial: M. Angeles Caamaño, Natalia Català,
M. Dolores Jiménez López, M. José Rodríguez Campillo.

Gemma Bel-Enguix per la Sèrie Linguistics, Biology and Computation,
i Esther Forgas per la Sèrie Español Lengua Extranjera.

Edita: Publicacions URV

ISSN: 2013-939X

ISBN: en tràmit

DL: T-1492-2010

Per a més informació de la revista consulteu la pàgina
<http://revista-triangle.blogspot.com/>

Publicacions de la Universitat Rovira i Virgili:

Av. Catalunya, 35 - 43002 Tarragona

Tel. 977 558 474 - Fax: 977 558 393

www.urv.cat/publicacions

publicacions@urv.cat

Arola Editors: Polígon Francolí, parcel·la 3, nau 5 - 43006 Tarragona

Tel. 977 553 707 - Fax 977 542 721

arola@arolaeditors.com

Cossetània Edicions: C. de la Violeta, 6 - 43800 Valls

Tel. 977 602 591 - Fax 977 614 357

cossetania@cossetania.com

Aquesta obra està subjecta a una llicència Attribution-NonCommercial-NoDerivs 3.0 Unported de Creative Commons. Per veure'n una còpia, visiteu <http://creativecommons.org/licenses/by-nc-nd/3.0/> o envieu una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contents

An Introduction to Grammatical Inference for Linguists

<i>Leonor Becerra-Bonache</i>	1
1 Motivation	1
2 Basic definitions	4
3 Formal models in Grammatical Inference	5
3.1 Gold: Identification in the limit	6
3.2 Angluin: Query Learning	8
The L^* algorithm	9
Running example	11
3.3 Linguistic discussion of these models	15
4 Towards a new formal model of language learning	17
4.1 What class of formal languages?	18
4.2 What source of data?	20
5 New proposals	23
5.1 Learning Simple External Contextual Languages	23
5.2 Learning from Positive Data and Corrections	24
6 Final Remarks	26
References	27

Logic Programming for Linguistics: A short introduction to Prolog, and Logic Grammars with Constraints as an easy way to Syntax and Semantics

<i>Henning Christiansen</i>	31
1 Prolog: Programming without programming	33
1.1 Prolog, lesson 1: A program as a knowledge base of facts	33

1.2	Prolog, lesson 2: Using variables to combine information and writing rules	36
1.3	Prolog, lesson 3: The rest of Prolog, with a focus on lists	38
1.4	More reading	40
2	Definite Clause Grammars	41
2.1	DCG, lesson 1: The basic grammar notation and syntax analysis	41
2.2	DCG, lesson 2: Adding features	42
2.3	More reading	44
3	A brief introduction to Constraint Handling Rules, CHR, and their application for abductive reasoning	44
3.1	Deduction, abduction, and induction in logic programming ..	44
3.2	Introducing Constraint Handling Rules by examples of abduction	46
3.3	Details of CHR	49
3.4	More reading	53
4	Language interpretation as abduction in Prolog+CHR	54
4.1	Introducing abductive interpretation by examples	54
4.2	More reading	58
5	One step further: Hyprolog	58
5.1	Assumptions: Like abduction but with time	58
5.2	Sketch of an example Hyprolog program	59
5.3	More reading	61
6	A few Prolog and CHR systems	61
	References	62

An Introduction to Natural Language Processing: the Main Problems

	<i>Veronica Dahl</i>	65
1	Definition, Scope	65
2	The most common problem in NLP	68
3	Divide and Reign	75
4	Concluding remarks	77
	References	78



An Introduction to Grammatical Inference for Linguists

Leonor Becerra-Bonache*

Research Group on Mathematical Linguistics
Rovira i Virgili University, Spain.

1 Motivation

This paper is meant to be an introductory guide to Grammatical Inference (GI), i.e., the study of machine learning of formal languages. It is designed for non-specialists in Computer Science, but with a special interest in language learning. It covers basic concepts and models developed in the framework of GI, and tries to point out the relevance of these studies for natural language acquisition.

How do children acquire their native language? This question has attracted the attention of researchers from different areas, including linguistics, cognitive science and computer science. Traditionally, this question has been addressed by linguists and psychologists. Their approach has specially been focused on making experiments with children that are acquiring their native language, with the ultimate goal of describing the process of natural language acquisition. There are basically two different kinds of studies: longitudinal studies (which focus on one child and collect data regularly to create extensive databases that can be found at CHILDES:

* Supported by a Marie Curie International Fellowship within the 6th European Community Framework Programme.

<http://childes.psy.cmu.edu/>); transversal studies (experiments are made with a group of children of different ages. Researchers try to test a hypothesis and design specific tasks that have to be performed by the children). Although important results have been obtained from all these studies, there are still many open questions about how children acquire their native language. This is why researchers have tried to approach the problem from a more interdisciplinary point of view, including such different scientific disciplines as Computer Science.

Within the field of Computer Science, *Artificial Intelligence* aims to study and design intelligent machines. This field was founded in the middle of the 50s. It has two different purposes:

One is to use the power of computers to augment human thinking, just as we use motors to augment human or horse power. Robotics and expert systems are major branches of that. The other is to use a computer's artificial intelligence to understand how humans think. In a humanoid way. If you test your programs not merely by what they can accomplish, but how they accomplish it, then you're really doing cognitive science; you're using Artificial Intelligence to understand the human mind. [34].

The founders of Artificial Intelligence were very optimistic about the future of this new field. For example, in 1965, H. Simon predicted that "... machines will be capable, within twenty years, of doing any work a man can do" [33]. Although important advances have been made in the last 45 years, this prediction has not come true yet. We have machines that can do "some of the things" that a man can do; for example, play soccer (<http://www.robocup.org/>), play some instruments (like Toyota's violin-playing robot), express feelings by moving their faces (like the MIT's robots: MDS and Kismet). Nevertheless, so far machines have been unable to *learn to speak*. The advantages of having a machine that can learn and speak a natural language would be innumerable. From a theoretical point of view, for example, we could better understand the process of natural language acquisition. From a practical point of view, to have a machine that is able to speak would definitely facilitate communication between humans and machines.

Within the field of Artificial Intelligence, *Machine Learning* aims to develop techniques that allow computers to learn. Machine Learning is concerned with the design and development of algorithms that allow computers to use data to change their behavior (an algorithm is a finite sequence



of instructions specifying how to solve a particular problem). Some of the Machine Learning applications are: natural language processing, search engines, medical diagnosis, detection of credit card fraud, classification of DNA sequences, speech and handwriting recognition, etc.

Grammatical Inference is a specialized subfield of Machine Learning that deals with the learning of formal languages from a set of examples. The basic framework can be regarded as a game played between two players: a teacher and a learner. The teacher provides data to the learner, and the learner (or learning algorithm), from these data, must identify the underlying language. For example, imagine that the target language (i.e., the language to be learnt) is ab^+ (i.e., a language that contains strings starting with one a , followed by at least one b). The teacher could provide the learner with strings that belong to the language (i.e., positive data), such as ab , abb , $abbb$... The learner uses this information to infer that the target language is ab^+ .

As we can see, this process has some similarities with the process of natural language acquisition; instead of a teacher we could have an adult, and instead of a learner, a child. Therefore, GI provides a good theoretical framework to study the problem of natural language acquisition. In fact, the initial theoretical foundations of GI were given by E.M. Gold, who was primarily motivated by the problem of children's language acquisition.

It is worth noting that the theory of formal languages was born in the 50's as a tool to describe natural language syntax. Hence, formal languages are an important tool to study natural languages. Moreover, formal results are also of great interest, because as A. Clark [16] pointed out:

Positive results can help us to understand how humans might learn languages by outlining the class of algorithms that might be used by humans, considered as computational systems at a suitable abstract level. Conversely, negative results might be helpful if they could demonstrate that no algorithms of a certain class could perform the task. In this case we could know that the human child learns his language in some other way [16, p. 26].

Therefore, by applying Grammatical Inference to the study of natural language acquisition, we could provide a formal model that explains how children acquire their native language. The study and development of a formal model of language learning is of great relevance, not only to better understand the process of natural language acquisition, but also for the



practical applications that such a model could have (for example, communication between humans and machines could be improved).

The remainder of the paper is organized as follows. We give some basic definitions in Section 2. In Section 3, we review some of the most important formal models investigated in GI, and we analyze them from a linguistic point of view. In section 4, we try to answer the following two questions: what classes of formal languages are interesting from a linguistic point of view? and what source of data should we provide our learning algorithm? In Section 5 we present some new lines of research in GI, motivated by studies of children's language acquisition. Concluding remarks are presented in Section 6.

2 Basic definitions

Formal languages are defined with respect to a given *alphabet*. The alphabet is a finite set of symbols, denoted Σ (e.g., $\Sigma = \{a, b\}$). A finite sequence of symbols chosen from some alphabet is called a *string* (e.g., $a, b, aa, ab, ba, bb, aaa...$). A *language* is a set of strings; among all the possible strings, some of them belong to the language and others do not (e.g., $ab, abb, abbb$ belong to the language ab^+ , but $a, ba, abba$ do not). A *grammar* is a finite mechanism that generates the elements of the language.

The Chomsky grammars are particular cases of *rewriting systems*, where the operation used to process the strings is rewriting (the replacement of a "short" substring of the processed string by another short substring). According to the form of their rules, the Chomsky grammars are classified as follows (from less to more expressive power): regular (REG), context-free (CF), context-sensitive (CS), recursively enumerable (RE). We call this the *Chomsky hierarchy* (see Figure 1). It is worth noting that Chomsky defined these formal grammars/languages with the ultimate goal of modeling the syntax of natural language.

For example, the language ab^+ is a regular language generated by the following regular grammar: $S \rightarrow aB, B \rightarrow b, B \rightarrow bB$.

Automata are recognizer devices that are able to decide whether or not an input string belongs to a specified language. The five basic families of languages in the Chomsky Hierarchy are also characterized by recognizing automata. These automata are: the finite automaton, the one-turn pushdown automaton, the pushdown automaton, the linearly bounded automaton, and the Turing machine, respectively.



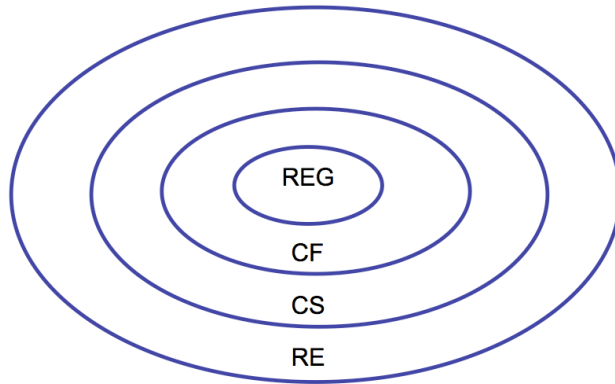


Fig. 1. The Chomsky Hierarchy

A *finite automaton* consists of a finite set of states, a finite alphabet of input symbols, and a set of transition rules. If the next state is always uniquely determined by the current state and the current input symbol, we say that the automaton is *deterministic*. Formally, a deterministic finite automata (DFA) is defined as a 5-tuple $(\Sigma, Q, \delta, q_0, F)$ where: Σ is the alphabet, Q is a finite set of states, T is the transition function ($T : Q \times \Sigma \rightarrow Q$, that is, from one state and reading a given symbol from the alphabet, we go to another state), q_0 is the initial state, and F the set of final states ($F \subseteq q$). A DFA takes a string as an input, and for each input symbol go to a state by following the transition function. When the last symbol is processed, depending on whether the DFA is in an accepting state or not, the string is accepted or rejected. A DFA characterizes the family of languages REG. See Figure 2 for an example of a DFA; initial state is marked with the symbol $>$ and the final (or accepted) state is marked with a double circle.

3 Formal models in Grammatical Inference

In this section we present two of the most important formal models developed within the field of GI. We also discuss some linguistics aspects of these models.



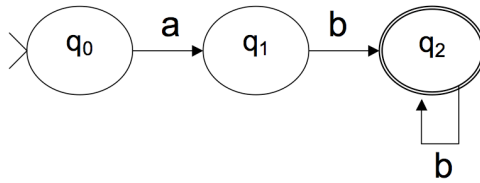


Fig. 2. Example of a Deterministic Finite Automata (DFA). This DFA recognizes the language ab^+ .

3.1 Gold: Identification in the limit

In 1967, Gold [21] introduced the model of *identification in the limit*. His final goal was to explain the acquisition of natural languages.

The study of language identification described here derives its motivation from artificial intelligence. The results and the methods used also have implications in computational linguistics, in particular the construction of discovery procedures, and in psycholinguistics, in particular the study of child learning (...).

I wish to construct a precise model for the intuitive notion “able to speak a language” in order to be able to investigate theoretically how it can be achieved artificially. Since we cannot explicitly write down the rules of English which we require one to know before we say he can “speak English”, an artificial intelligence which is designed to speak English will have to learn its rules from implicit information. That is, its information will consist of examples of the use of English and/or of an informant who can state whether a given usage satisfies certain rules of English, but cannot state these rules explicitly. [21, pp. 447–448].

Identification in the limit views learning as an infinite process. In this model, the learner passively receives more and more examples, and has to produce a hypothesis of the target language. If the learner receives new examples that are not consistent with his hypothesis, he has to change it. His hypothesis has to converge to a correct hypothesis. We say that the learner identifies the target language in the limit if, after a finite number of examples, he makes a correct guess and does not alter his guess thereafter.

It is worth noting that under this criterion, the learner cannot be certain of having correctly guessed the target language, since he may receive new



examples that are not consistent with his hypothesis. Gold justifies the study of identifiability in the limit in the following way:

My justification for studying identifiability in the limit is this: A person does not know when he is speaking a language correctly; there is always the possibility that he will find that his grammar contains an error. But we can guarantee that a child will eventually learn a natural language, even if it will not know when it is correct. [21, p. 450].

Gold studied two different learning settings: i) Learning from *text*: the learner only receives positive data (strings that belong to the language); ii) Learning from *informant*: the learner receives positive and negative data (i.e., strings that belong to the language and strings that do not).

Gold proved that *superfinite* classes of languages (a class is superfinite if it contains all finite languages and at least one infinite language) cannot be identified in the limit from only positive data. This implies that none of the classes of languages defined by Chomsky to model natural language syntax is identifiable in the limit from only positive data. Therefore, the following question arises: How do children overcome this theoretical hurdle? Gold suggested the following hypothesis:

If one accepts identification in the limit as a model of learnability, then this conflict must lead to at least one of the following conclusions:

1. *The class of possible natural languages is much smaller than one would expect from our present models of syntax. That is, even if English is context-sensitive, it is not true that any context-sensitive language can occur naturally. Equivalently, we may say that the child starts out with more information than that the language it will be presented is context-sensitive. In particular, the results on learnability from text imply the following: The class of possible natural languages, if it contains languages of infinite cardinality, cannot contain all languages of finite cardinality.*
2. *The child receives negative instances by being corrected in a way we do not recognize. If we can assume that the child receives both positive and negative instances, then it is being presented information by an "informant". The class of primitive recursive languages, which includes the class of context-sensitive languages, is identifiable in the limit from an informant. The child may receive the equivalent of negative instances for the purpose of grammar acquisition when it does not get the desired response to an utterance. It is difficult to interpret the actual training*



program of a child in terms of the naive model of a language assumed here.

3. *There is an a priori restriction on the class of texts which can occur, such as a restriction on the order of text presentation. The child may learn that a certain string is not acceptable by the fact that it never occurs in a certain context. This would constitute a negative instance. [21, p. 453–454].*

Studies along these lines have shown that the first path (the class of potential natural language is more restrictive than those defined by Chomsky) can be successful (see, [1,25,31]). In linguistics, it is also generally assumed that the first conclusion holds.

Now it seems evident to many linguists (notably, Chomsky [40,43]) that children are not genetically prepared to acquire any arbitrary language on the basis of the kind of casual linguistic exposure typically afforded the young. Instead, a relatively small class \mathcal{H} of languages may be singled out as “humanly possible” on the basis of their amenability to acquisition by children, and it falls to the science of linguistics to propose a nontrivial description of \mathcal{H} [23, p.29].

3.2 Angluin: Query Learning

D. Angluin introduced the query learning model in [2]. In this model, the learner is allowed to make queries to the teacher. The teacher (or oracle) knows the target language and answers the queries made by the learner correctly (he is perfect).

The learner (or learning algorithm) can only make queries from a given set. After asking a finite number of questions, the learner must return a hypothesis. The learner’s hypothesis has to be the correct one (that is why this kind of learning is also known as *exact learning*).

There are different kinds of queries available to the learner, but just two of them have established themselves as the standard combination to be used:

- *Membership queries* (MQs): the learner asks if a string w is in the language, and the teacher answers “yes” if w belongs to the target language, and “no” otherwise.
- *Equivalence queries* (EQs): the learner asks if his hypothesis H is correct, and the teacher answers “yes” if H is equivalent to the target language



L and “no” otherwise. If the answer is “no”, a counterexample x is returned (i.e., a string in the symmetric difference of H and L).

A teacher that can answer MQs and EQs is called a MAT teacher (minimally adequate teacher). In [2], Angluin gave an algorithm known as L^* , which learns DFA from MAT. She proved that it is possible to learn DFA from MQs and EQs in polynomial time, and it was conjectured that richer classes than DFA cannot be inferred through a polynomial use of MAT. Since then, the L^* algorithm has become the main reference and one of the most relevant results in the framework of learning from queries. Below we briefly review the learning algorithm L^* . Details can be found in [2].

The L^* algorithm

The general idea of the algorithm is to repeat the following loop until the answer to an EQ is “yes”:

- Find a closed and consistent observation table (representing a DFA) by means of MQs
- Ask an EQ
- If the answer is “no” (it is not the correct acceptor), then use the counterexample to update the table

What is an observation table? The information during the learning process is organized in a table called *observation table*. An observation table is a two-dimensional table, with both rows and columns indexed by strings (for example, see Figure 3).

We can differentiate three main parts in an observation table:

- S : a prefix-closed set of strings. Rows labeled by elements of S are the candidates for states of the automaton being constructed.
- T : in this part of the table we find rows labeled by elements of $S \cdot \Sigma$ (i.e., elements of S concatenated with all the symbols of the alphabet). These rows are used to construct the transition function.
- E : a suffix-closed set of strings. Columns labeled by elements of E correspond to distinguishing experiments for these states.

The observation table will be denoted (S, E, T) . By concatenating the string of a row r with the string of a column c we get a string rc . If the string rc is in the language, the corresponding cell contains a 1, and 0 otherwise.



		λ	a	← Experiments (E)
States (S) →	λ	1	0	
	a	0	0	
Transitions (T) →	b	1	0	
	aa	0	0	
	ab	1	0	

Fig. 3. Observation table. $\Sigma = \{a, b\}$

An observation table is called *closed* if any row of $S \cdot \Sigma$ corresponds with some row in S . An observation table is called *consistent* if every equivalent pair of rows in S remains equivalent after appending any symbol. When we have a closed and consistent table we can build the corresponding DFA and make an EQ.

How do we build a DFA? The L^* algorithm uses the observation table to build one. We define a corresponding automaton $A(S, E, T)$ over the alphabet Σ , with state set Q , initial state q_0 , accepting states F , and transition function δ as follows:

- $Q = \{row(s) | s \in S\}$
- $q_0 = row(\lambda)$
- $F = \{row(s) | s \in S \text{ and } T(s) = 1\}$
- $\delta(row(s), a) = row(s \cdot a)$

For example, as the reader can easily verify, the observation table depicted in Figure 3 is closed and consistent. So, we can construct a DFA from this table. There are only two candidates for states: the row labeled λ and the row labeled a . The first contains 10 and the second 00; these values can be considered as a codification of the state. Therefore, we can call q_0 all the rows that have the value 10, and q_1 all the rows with the value 00. Now, by using the other rows, we know that: from q_0 , by reading the symbol a , we go to state q_1 (value of the row labelled a), and by reading the symbol b , we go to state q_0 (value of the row labelled b); from q_1 , by reading the symbol a we go to state q_1 (value of the row labelled aa), and by reading b we go to



state q_0 (value of the row labelled ab). Moreover, q_0 is both initial and final state. In this way, we can construct the corresponding automaton, which is depicted in Figure 4.

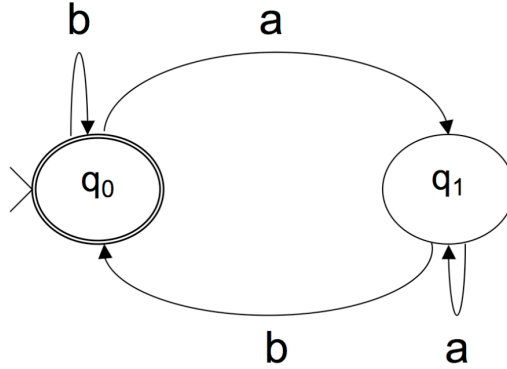


Fig. 4. Automaton corresponding to the observation table depicted in Figure 3

After making the EQ, if the conjectured DFA is the correct one, we will get a positive answer from the teacher. Then, the algorithm halts. If the conjectured DFA is not the correct one, we will get a counterexample. In such a case, we have to: i) Add the counterexample and all its prefixes to S ; ii) Update the table using MQs for missing elements. We shall explain all these steps in greater detail using an example.

Running example

Let the alphabet $\Sigma = \{0, 1\}$, and a language $L = (0 + 110)^+$. The minimal automaton associated with the mentioned language is shown in Figure 5.

Initially the learner starts with the following observation table described as Table 1.

This table is not closed because $row(0)$ does not belong to $rows(S)$. L^* chooses to add the string 0 to S , 00 and 01 to $S\Sigma - S$, and then queries 00 and 01 to construct the observation table T_2 shown in Table 2.



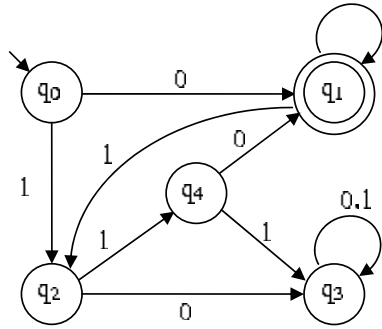


Fig. 5. Minimal automaton associated to the language $L_1 = (0 + 110)^+$

Table 1. $S = \{\lambda\}, E = \{\lambda\}$

T_1	λ
λ	0
0	1
1	0

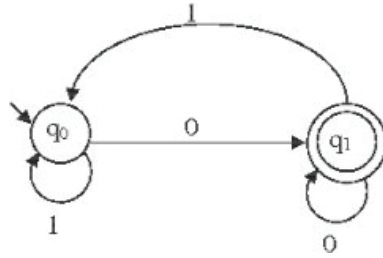
Table 2. $S = \{\lambda, 0\}, E = \{\lambda\}$

T_2	λ
λ	0
0	1
1	0
00	1
01	0

This observation table is closed and consistent, so L^* makes a conjecture of the automaton A_1 , shown in Figure 6.

A_1 is not a correct automaton for L , so the teacher selects a counterexample. In this case we assume that the counterexample 10 is returned (it is not in L but accepted by A_1).

To process the counterexample 10, L^* adds the strings 1 and 10 to S (the string λ is already in S), and queries the strings 11, 100 and 101 to construct the observation table T_3 shown in Table 3.

Fig. 6. Associated automaton: A_1 Table 3. $S = \{\lambda, 0, 1, 10\}$, $E = \{\lambda\}$

T_3	λ
λ	0
0	1
1	0
10	0
00	1
01	0
11	0
100	0
101	0

This observation table is closed, but not consistent since $row(\lambda) = row(1)$ but $row(0) \neq row(10)$. Thus L^* adds the string 0 to E , and queries the strings required to construct the observation table T_4 shown in Table 4.

Table 4. $S = \{\lambda, 0, 1, 10\}$, $E = \{\lambda, 0\}$

T_4	λ	0
λ	0	1
0	1	1
1	0	0
10	0	0
00	1	1
01	0	0
11	0	1
100	0	0
101	0	0



This observation table is closed, but not consistent since $row(1) = row(10)$ but $row(11) \neq row(101)$. Thus L^* adds the string 10 to E , and queries the strings required to construct the observation table T_5 shown in Table 5.

Table 5. $S = \{\lambda, 0, 1, 10\}$, $E = \{\lambda, 0, 10\}$

T_5	λ	0	10
λ	0	1	0
0	1	1	0
1	0	0	1
10	0	0	0
00	1	1	0
01	0	0	1
11	0	1	0
100	0	0	0
101	0	0	0

This observation table is closed and consistent, so L^* conjectures the automaton A_2 shown in Figure 7.

A_2 is not a correct acceptor for L , so the teacher answers the conjecture with a counterexample. We assume that the counterexample supplied is 11110, which is not in L but is accepted by A_2 .

L^* adds the counterexample and all its prefixes to S and constructs the observation table T_6 shown in Table 6.

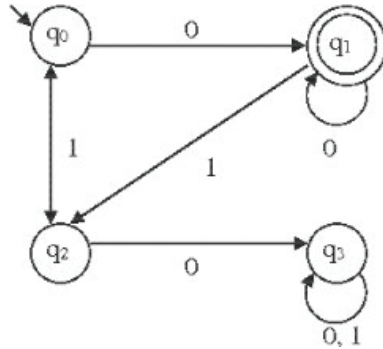
This table is found to be closed but not consistent, since $row(\lambda) = row(11)$ but $row(1) \neq row(111)$.

Thus L^* adds the string 110 to E and queries the necessary strings to construct the observation table T_7 shown in Table 7.

This table is closed and consistent. The automaton conjectured by L^* now corresponds to the correct acceptor for the language L , so the Teacher replies to this conjecture with *yes* and L^* terminates with this automaton as its output.

The total number of queries during this run of L^* is 3 EQs (the last one was successful) and 44 MQs.



Fig. 7. Associated automaton: A_2 **Table 6.**

$S = \{\lambda, 0, 1, 10, 11, 111, 1111, 11110\}$,
 $E = \{\lambda, 0, 10\}$

T_6	λ	0	10
λ	0	1	0
0	1	1	0
1	0	0	1
10	0	0	0
11	0	1	0
111	0	0	0
1111	0	0	0
11110	0	0	0
00	1	1	0
01	0	0	1
100	0	0	0
101	0	0	0
110	1	1	0
1110	0	0	0
11111	0	0	0
111100	0	0	0
111101	0	0	0

Table 7.

$S = \{\lambda, 0, 1, 10, 11, 111, 1111, 11110\}$,
 $E = \{\lambda, 0, 10, 110\}$

T_7	λ	0	10	110
λ	0	1	0	1
0	1	1	0	1
1	0	0	1	0
10	0	0	0	0
11	0	1	0	0
111	0	0	0	0
1111	0	0	0	0
11110	0	0	0	0
00	1	1	0	1
01	0	0	1	0
100	0	0	0	0
101	0	0	0	0
110	1	1	0	1
1110	0	0	0	0
11111	0	0	0	0
111100	0	0	0	0
111101	0	0	0	0

3.3 Linguistic discussion of these models

The formal models presented in the previous section are based on different learning settings (i.e., the type of data used in the learning process and



the way in which these data are provided to the learner is different in both cases) and different criteria for a successful inference (i.e., the conditions under which we say that a learner has been successful in the language learning task are different). But, which one is better for modeling natural language acquisition? Below we review some of the accepted and controversial aspects of these models.

We can find some similarities between learning in Gold's model and first language acquisition. In both cases there is a process of *improvement*: in identification in the limit model, the new conjecture is better than the previous guess; in the case of first language acquisition there is a progressive improvement of the language acquired by the child. However, there are some aspects of Gold's model that are controversial from a linguistic point of view. For example:

- *In the limit* denotes the criterion of success, which assumes that there is no limit on how long it can take the learner to guess the correct language. Hence, considerations of efficiency form a somewhat separate line of analysis from Gold's work, which was concerned with limiting behavior rather than speed of learning. However, from the natural language acquisition point of view efficiency is also important. Although learning a natural language is an infinite process, we are able to learn the language in an efficient way.
- The learner passively receives strings of the language. However, we know that natural language learning is more than that: children also interact with their environment.
- The current hypothesis has to be consistent with all the examples seen so far. Moreover, the learner hypothesizes complete grammars instantaneously. From a linguistic point of view these assumptions are unrealistic (e.g., children are unlikely to remember the entire record of sentences ever addressed to them).

Therefore, the definition of identification in the limit postulates greatly idealized conditions, as compared to the conditions under which children learn language.

Angluin's model addresses an important tool available to a child, i.e., queries to a teacher (usually, a family adult member). Therefore, the query learning model might be useful when representing several aspects of the process of children's language acquisition. However, this model also has some controversial aspects from a linguistic point of view:



- The type of queries introduced with this model are quite un-natural for real learning environments. For example, an equivalency query will never be produced in a real situation; a child will never ask an adult if his grammar is correct.
- The learner does not really interact with the teacher; he can ask MQs or EQs, but he does not really communicate with the teacher by producing sentences, etc. In the communication between children and adults we can see that the role of the children is more active, and not limited to asking this kind of queries.
- Angluin's model is known as exact learning. However, from a linguistic point of view, everybody has (small) imperfections in their linguistic competence.
- The teacher in this model is assumed to know everything and always gives the correct answers. Therefore, he is an ideal teacher, which does not correspond with a real situation.

The third model studied in GI is called the PAC learning model (probably approximately correct), which was introduced by Valiant in [35]. It is a probabilistic model of learning from random examples; the distribution over the examples is unknown, and the examples are sampled under this distribution. The learner is required to be able to learn from this sample and under any probability distribution, but exactitude is not required (a small error is permitted since one may be unlucky during the sampling processes). Taking into account that exact learning is too hard in a real context, approximate learning could be a good way of dealing with children's language acquisition. However, the requirement that the examples have the same distribution throughout the process is too strong for practical situations.

As we have seen, all these models have aspects that make them suitable for studying natural language acquisition to a certain extent, but other aspects of the models make them unsuitable for this task. Therefore, we can conclude that none of these models perfectly accounts for natural language acquisition.

4 Towards a new formal model of language learning

As we have pointed out, the study and development of formal models of language learning is of great interest if we are to better understand the pro-



cess of natural language acquisition. In the section above we have seen that the models that have been proposed so far in GI have many controversial aspects from a linguistic point of view. Part of the reason is because GI studies have been specially focused on obtaining formal results, and they have been more interested in the mathematical aspects of the models than in their linguistic relevance.

Therefore, it would be interesting to develop new formal models of language learning that take greater account of studies of natural language acquisition (in this way, we could avoid some of the controversial aspects of the models proposed so far). In order to do this, it is important to address two questions: what classes of formal languages are interesting from a linguistic point of view?; what source of data should we provide our learning algorithm with? We try to answer these questions below.

4.1 What class of formal languages?

The theory of formal languages arose born in the second half of the 20th century as a tool to describe natural language syntax. As we have pointed out, the goal of GI studies is to learn formal languages from data. Most research into GI has focused on learning two classes of formal languages: regular and context-free languages (two of the classes with least generative power in the Chomsky hierarchy). However, what class of formal languages is more interesting from a linguistic point of view?

In order to answer this question, first, we need to answer the following question: Where are natural languages located in the Chomsky hierarchy? This question has been a subject of debate for a long time. This debate was focused on trying to determine whether natural languages are CF or not. In the late 80s, examples of structures that are not CF were discovered in several natural languages. Here are some examples of such constructions:

- **Dutch:** Bresnan et al. studied cross-serial dependencies in Dutch, arguing against the context-freeness of natural language.

While Dutch may or may not be CF in the weak sense, it is not strongly CF: there is no CFG that can assign the correct structural descriptions to Dutch cross-serial dependency constructions. [13, p. 314]

The following example shows a duplication-like structure $\{w\bar{w} \mid w \in \{a,b\}^*\}$, where \bar{w} is the word obtained from w by replacing each letter with its barred copy.



...dat Jan Piet Marie de Kinderen zag helpen laten zwemmen
(That Jan saw Piet help Marie make the children swim)

This is only *weakly* non-context-free, i.e., only in the deep structure.

- **Bambara:** Bambara, an African language of the Mande family, was studied by Culy in [19]. He provided another argument against context-freeness based on the morphology of words in that language.

In this paper I look at the possibility of considering the vocabulary of a natural language as a sort of language itself. In particular, I study the weak generative capacity of the vocabulary of Bambara, and show that the vocabulary is not context-free. This result has important ramifications for the theory of syntax of natural language. [19, p. 349].

A duplication structure is found in the vocabulary of Bambara, demonstrating a strong non-context-freeness, i.e., on the surface and in the deep structure:

malonyininafilèla o malonyininafilèla o
(one who searches for rice watchers + one who searches for
rice watchers = whoever searches for rice watchers)

This has the structure $\{wcw \mid w \in \{a, b\}^*\}$. But also the *crossed agreement* structure $\{a^n b^m c^n d^m \mid m, n > 0\}$ can be inferred.

- **Swiss German:** The paper by Shieber [32], offers evidence for the non-context-freeness of natural language. He collected data from native Swiss German speakers, and provided a formal proof of the non-context-freeness of Swiss German.

Using a particular construction of Swiss German, the cross-serial subordinate clause, we have presented an argument providing evidence that natural languages can indeed cross the context-free barrier. The linguistic assumptions on which our proof rests are small in number and quite weak; most of the proof is purely formal. In fact, the argument would still hold even if Swiss German were significantly different from the way it actually is, i.e., allowing many more constituent orders, cases and constructions, and even if the meanings of the sentences were completely different. [32, p. 330].



The following example is a strong non-context-free structure, again showing crossed agreement:

*Jan säit das mer (d'chind)^m (em Hans)ⁿ es huus haend wele (laa)^m
(hälfe)ⁿ aastriiche*

(Jan said that we wanted to let the children help Hans paint the house)

This has the structure $xwa^mb^ncy^md^nz$, where a, b stand for accusative, dative noun phrases, respectively, and c, d for the corresponding accusative, dative verb phrases, respectively.

So, all these studies provide a negative answer to the question of whether natural languages are CF or not. Moreover, they suggest that natural languages can only be described by a generative capacity that is greater than context-free grammar. But, how much power is needed to describe these non-CF constructions?

In 1985, Joshi [24] introduced the notion of the *Mildly Context-Sensitive* family of languages. The general idea was to provide a device that was able to generate CF and non-CF structures, but keep the generative power under control. There are very well known mechanisms for fabricating MCS families: for example, tree adjoining grammars, head grammars, combinatory categorial grammars. In the Chomsky hierarchy they are somewhere between CF and CS. However, is it necessary for such formalisms to generate all CF languages? We can find natural language constructions that are neither REG nor CF, and also some REG or CF constructions that do not appear naturally in sentences. Therefore, as some authors point out [7, 26, 27], natural languages could occupy an orthogonal position in the Chomsky hierarchy.

So, it would be desirable to find new formalisms that have the following two properties: i) They are able to generate Mildly Context-sensitive languages (i.e., they generate multiple agreement, crossed agreement and duplication structures, and they are computational feasible); ii) They occupy an orthogonal position in the Chomsky hierarchy (i.e., they contain some REG, some CF, and so on).

4.2 What source of data?

The learning paradigms that have most been studied in GI are: learning from positive data (most of them), and learning from queries. However, if



we want to correctly simulate natural language learning, we should provide our learning algorithm with the same kind of examples that are available to a child. But one of the questions that is still a subject of debate in Linguistics is precisely this: what source of data is available to children during the learning process?

It is widely accepted that children receive positive data; that is, sentences that are grammatically correct. However, the availability of another kind of data (called negative data) is still a matter of substantial controversy. Do children receive negative data and use them during the learning process?

There have been three main responses to this question. The first proposal is that children do not receive negative data and they must rely on innate information to acquire their native language. This proposal is based on the *poverty of stimulus* argument: there are principles of grammar that cannot be learnt from only positive data, and since children do not receive negative data (i.e., evidence about what is not grammatical), one can conclude that the innate linguistic capacity is what provides the additional knowledge that is necessary for language learning. Further justification for innateness was drawn from Gold's negative result on learning from positive data. Moreover, Brown and Hanlon [14] analyzed adult approval and disapproval of child utterances (for example, adult's answers such as "That's right", "Correct", "That's wrong", "No"). They found no relation between this type of answer and the grammaticality of the sentences produced by the children, and this was also taken to show that children do not receive negative data. However, it is worth noting that parents do not usually address their children in this way. Should only explicit disapproval count as negative evidence? Do adults correct children in a different way?

The second proposal is that children receive negative data in the form of *different reply-types* given in response to grammatical versus ungrammatical child utterances. Hirsh-Pasek et al. [22], Demetras et al. [20], and Morgan and Travis [29] proposed that parents respond to ungrammatical child utterances by using different types of answers from those they use when responding to grammatical utterances. Under this view, the reply type would indicate to the child whether an utterance was grammatically correct or not. For example, if parents tend to respond with an expansion when the child's utterance is incorrect, but repeat the sentences that are grammatically correct, then adult use of an expansion would signal that the child's utterance was incorrect. However, Marcus [28] analyzed all these studies and concluded that there is no evidence that this kind of data is necessary to learn a



language or even that they exist. Even if they exist, a child would learn what utterances are correct only after complex statistical comparisons. Therefore, these results were also used to show that internal mechanisms are necessary to explain how children get rid of errors to acquire their native language.

The third proposal is that children receive negative evidence in the form of *reformulations*, and not only do they detect them, they also make use of the information. Chouinard and Clark [15] proposed this new view of negative evidence. They consider that the reply-types proposal does not take into account if the adult's answer contains corrective information (then, answers that are corrective are grouped with those that are not). Therefore, if only the reply-type is taken into account, it could be difficult to identify the error made. On the basis of Clark's theory of contrast [17, 18], Chouinard and Clark proposed adult reformulations as negative evidence. They consider that it is in the to-and-fro of conversation that children receive information about whether their utterances are appropriate for their intended meanings. For example (extracted from CHILDES database, Kuczaj):

Abe: milk milk

Father: you want milk?

Abe: uh-huh

Father: Ok. Just a second and I'll get you some.

In this conversation, Abe is about two years and a half. She produces an incorrect sentence and, immediately after, the father reformulates her sentence by checking on what the child had intended to say. After that, the child acknowledges the reformulation. Therefore, as we can see: i) Adult correction preserves the same meaning of the child; ii) Adult uses the correction to keep the conversation on track (adult reformulates the sentence just to make sure that he has understood the child's intentions); iii) Child utterance and adult correction have the same meaning, but different form. Chouinard and Clark analyzed longitudinal data from five children between two and four years old, and they showed that adults reformulate erroneous child utterances often enough to help learning. Moreover, they showed that children not only detect differences between their own utterance and the adult reformulation, they also make use of the information.

Do corrections give positive or negative information? As we can see, these types of corrections contain positive and negative information at the same time. On the one hand, corrections are positive data, since a correction



is a sentence that is grammatically correct. On the other hand, they also give us negative information; as Chouinard and Clark pointed out:

Since, like adults, children attend to contrast in form, any change in form that does not mark a distinct, different, meaning will signal to children that they may have produced something that is not acceptable in the target language. And this fits the classic definition of negative evidence [15, p. 666]

It is worth noting that during the first stages of children's language acquisition, children receive corrections that preserve the meaning of what they intend to convey. However, this kind of information has not been taken into account in formal models of language learning. Why should it not be taken into account? What is the effect of corrections on the process of language learning? A model that takes corrections into account could allow us to answer this question.

5 New proposals

As we have seen, REG and CF languages have a limited expressive power to describe some aspects of the syntax of natural languages. Moreover, corrections could play an important role during the process of language acquisition. Taking into account all these ideas, we shall briefly review two new lines of research that have been proposed in the last four years.

5.1 Learning Simple External Contextual Languages

We have pointed out in the section above that it would be desirable to have a mechanism that can generate MCS languages and occupy an orthogonal position in the Chomsky hierarchy. Becerra-Bonache [7] proposed and studied a non-classical mechanism that has these interesting properties: *Simple External Contextual* grammars (SEC).

A SEC produces a language starting from a string called *base*, and iteratively adding contexts (i.e., pair of strings) at the ends of the current string. Formally, a SEC grammar is defined as $G = (\Sigma, B, C)$, where:

- Σ : alphabet.
- B : one p -word (i.e., a p -dimensional vector whose components are words/ strings) over Σ , called the base of the grammar.



- C : a finite set of p -contexts (i.e., a p -dimensional vector whose components are contexts) over Σ , called the set of contexts of G .

Here is an example. Let us assume we have a SEC grammar with 2 dimensions, where: $\Sigma = \{a, b, c\}$, $B = \{(\lambda, \lambda)\}$, and $C = \{c_1 = [(a, b), (c, \lambda)]\}$. Starting from the base (λ, λ) , if we apply the context once we obtain the 2-word $(a\lambda b, c\lambda\lambda) = (ab, c) = abc$. Now, starting from (ab, c) , if we again apply the context we obtain $(aa\lambda bb, cc\lambda\lambda\lambda) = (aabb, cc) = aabbcc$. Note that by using this grammar, we can generate the following non-CF language: $L = \{a^n b^n c^n \mid n \geq 0\}$. The generation process is depicted in Figure 8.

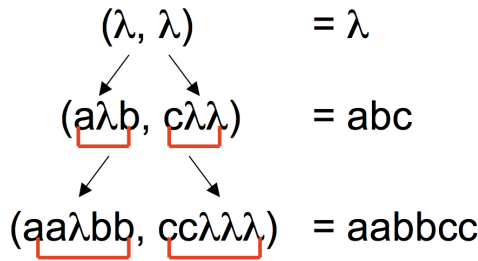


Fig. 8. Derivation process of the SEC grammar $G = (\Sigma = \{a, b, c\}, B = \{(\lambda, \lambda)\}, \text{ and } C = \{c_1 = [(a, b), (c, \lambda)]\})$

Becerra-Bonache [7] proved that SEC can generate MCS languages and occupies an orthogonal position in the Chomsky hierarchy (see Figure 9). Moreover, the learnability of SEC from positive data has been studied in [8, 12, 30].

5.2 Learning from Positive Data and Corrections

As we have seen in the section above, studies on children's language acquisition show that corrections are available to children. Although the main source of information received during the process of natural language acquisition is positive data, corrections could play a complementary role in the process. Therefore, it is of great interest to study the effects of corrections on language learning.

Taking all this into account, Becerra-Bonache [7] tried to apply the idea of corrections to GI studies, and more concretely to the query learning model



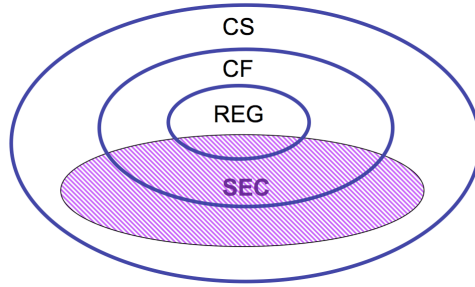


Fig. 9. Location of SEC in the Chomsky hierarchy; it is incomparable with REG and CF, but included in CS.

introduced by D. Angluin. In this way, Becerra-Bonache presented a new type of query called *correction query* (CQ); when a string is submitted to the oracle, either he validates it if it belongs to the target language, or he proposes a correction. The learnability of different classes of languages using CQs has been studied in [6,9–11]. The results obtained so far show that the concept of CQ generates new challenging results in the field of GI.

The kinds of correction considered in the papers cited above are mainly *syntactic corrections* based on proximity between strings. However, in natural situations, a child's erroneous utterance is corrected by parents on the basis of the meaning that the child intends to express (i.e., the correction *preserves* the intended meaning of the child's utterance). In [3–5], Angluin and Becerra-Bonache proposed a new computational model that gives an account of this kind of correction. The model takes into account the context, semantics, positive data and corrections. It includes two different tasks: comprehension and production.

It is worth noting that the model proposed by Angluin and Becerra-Bonache is mainly inspired by studies on children's language acquisition. In this new approach, the teacher is able to understand a flawed utterance provided by the learner and respond with a correct utterance for that meaning (by using meaning-preserving corrections). Moreover, the learner can recognize that the teacher's utterance has the same meaning but a different form. This model has allowed them to investigate aspects of the roles of semantics and corrections in the process of learning to understand and speak a natural language. The model has been tested with limited sublanguages of several natural languages, and the results show that access to semantics



facilitates language learning, and that the presence of corrections by the teacher has an effect on language learning by the learner (even if the learner does not give corrections any special treatment).

6 Final Remarks

Research in formal models of language acquisition is of great importance in understanding how children acquire their native language. Moreover, the simulation of this human capability could have important implications in the field of human language technologies; for example, if computers are able to learn a language like a human, the user could interact with a computer in a more natural way (without any special skill or training).

GI provides a good theoretical framework in which to develop such models. However, the models that have been proposed so far in this field, do not take into account important aspects of natural language acquisition, and, hence, do not give a good account of the process.

In this paper we have pointed out some of the weak points of these models, and we have proposed some new ideas (based on studies of children's language acquisition) that could be taken into account in new formal models of language learning. On the one hand, we consider that studies in GI should focus on classes of languages that have more important linguistic properties; that is, classes of languages that are mildly context-sensitive and occupy an orthogonal position in the Chomsky hierarchy (like natural languages are believed to have). An example of a class with such properties is the Simple External Contextual. On the other hand, we have seen that corrections can also play an important role in the process of natural language acquisition so it is also of interest to take into account this source of information in formal models of language learning. Studies along these lines show the challenging results that can be obtained when this idea is taken into account.

Therefore, with this paper we have intended to show that studying natural language acquisition from an interdisciplinary point of view can be of interest. Ideas and techniques from such different research areas as linguistics, cognitive science and computer science, could definitely help to find an answer to the question of how children acquire their native language.



References

1. D. Angluin. Inference of reversible languages. *Journal of the Association for Computing Machinery*, 29(3):741–765, 1982.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
3. D. Angluin and L. Becerra-Bonache. *A model of semantics and corrections in language learning*. YALEU/DCS/TR-1425, April, 2010.
4. D. Angluin and L. Becerra-Bonache. *Learning meaning before syntax*. YALEU/DCS/TR-1478, July, 2008.
5. D. Angluin and L. Becerra-Bonache. Learning meaning before syntax. In *ICGI*, pages 1–14. Springer-Verlag, Berlin, 2008.
6. D. Angluin, L. Becerra-Bonache, A.H. Dediu, and L. Reyzin. Learning finite automata using label queries. In *ALT*, pages 264–276. Springer-Verlag, Berlin, 2008.
7. L. Becerra-Bonache. *On the Learnability of Mildly Context-Sensitive Languages using Positive Data and Correction Queries*. PhD thesis, Rovira i Virgili University, 2006.
8. L. Becerra-Bonache, J. Case, S. Jain, and F. Stephan. Iterative learning of simple external contextual languages. *Theoretical Computer Science*, 411:2741–2756, 2010.
9. L. Becerra-Bonache, C. de la Higuera, J.C. Janodet, and F. Tantini. Learning balls of strings from edit corrections. *Journal of Machine Learning Research*, 9:1841–1870, 2008.
10. L. Becerra-Bonache and A. H. Dediu. Learning from a smarter teacher. In Emilio Corchado and Hujun Yin, editors, *IDEAL*, pages 200–207. Springer-Verlag, Berlin, 2009.
11. L. Becerra-Bonache, A. H. Dediu, and C. Tîrnuca. Learning dfa from correction and equivalence queries. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *ICGI*, volume 4201, pages 281–292. Springer-Verlag, Berlin, 2006.
12. L. Becerra-Bonache and T. Yokomori. Learning mild context-sensitiveness: Toward understanding children’s language learning. In G. Paliouras and Y. Sakakibara, editors, *ICGI*, volume 3264, pages 53–64. Springer-Verlag, Berlin, 2004.
13. J. Bresnan, R.M. Kaplan, S. Peters, and A. Zaenen. Cross-serial dependencies in dutch. In W.J. Savitch, E. Bach, W. Marsh, and G. Safran-Naveh, editors, *The Formal Complexity of Natural Language*, pages 286–319. D. Reidel, Dordrecht, 1987.
14. R. Brown and C. Hanlon. Derivational complexity and the order of acquisition in child speech. In J. R. Hayes, editor, *Cognition and the development of language*. Wiley, New York, 1970.



15. M.M. Chouinard and E.V. Clark. Adult reformulations of child errors as negative evidence. *Journal of Child Language*, 30:637–669, 2003.
16. A. Clark. Grammatical inference and first language acquisition. In *Workshop on Psychocomputational Models of Human Language Acquisition*, pages 25–32. Geneva, 2004.
17. E.V. Clark. The principle of contrast : a constraint on language acquisition. In B. MacWhinney, editor, *Mechanisms of language acquisition*. Erlbaum, Hillsdale, NJ, 1987.
18. E.V. Clark. *The lexicon in acquisition*. CUP, Cambridge, 1993.
19. C. Culy. The complexity of the vocabulary of bambara. In W.J. Savitch, E. Bach, W. Marsh, and G. Safran-Naveh, editors, *The Formal Complexity of Natural Language*, pages 349–357. 1987.
20. M. J. Demetras, K. N. Post, and C. E. Snow. Feedback to first language learners: the role of repetitions and clarification questions. *Journal of Child Language*, 13:275–292, 1986.
21. E.M. Gold. Language identification in the limit. *Information and Control*, 10:447–474., 1967.
22. K. Hirsh-Pasek, R. A. Treiman, and M. Schneiderman. Brown and hanlon revisited: mothers' sensitivity to ungrammatical forms. *Journal of Child Language*, 11:81–88, 1984.
23. S. Jain, D. Osherson, J.S. Royer, and A. Sharma. *Systems that Learn*. MIT Press, Cambridge, MA, 1999.
24. A. K. Joshi. How much context-sensitivity is required to provide reasonable structural descriptions: Tree adjoining grammars. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, pages 206–250. Cambridge University Press, New York, NY, 1985.
25. M. Kanazawa. *Learnable Classes of Categorical Grammars*. Cambridge University Press, New York, NY, 1998.
26. M. Kudlek, C. Martín-Vide, A. Mateescu, and V. Mitran. Contexts and the concept of mild context-sensitivity. *Linguistics and Philosophy*, 26(6):703–725, 2002.
27. A. Manaster-Ramer. Some uses and abuses of mathematics in linguistics. In C. Martín-Vide, editor, *Issues in Mathematical Linguistics*, pages 73–130. John Benjamins, Amsterdam, 1999.
28. G.F. Marcus. Negative evidence in language acquisition. *Cognition*, 46:53–95, 1993.
29. J. L. Morgan, K. L. Bonamo, and L. L. Travis. Negative evidence on negative evidence. *Developmental Psychology*, 31:180–197, 1989.
30. T. Oates, T. Armstrong, L. Becerra-Bonache, and M. Atamas. Inferring grammars for mildly context sensitive languages in polynomial-time. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *ICGI*, volume 4201, pages 137–147. Springer-Verlag, Berlin, 2006.



31. Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information Processing Letters*, 97:23–60, 1992.
32. S.M. Shieber. Evidence against the context-freeness of natural languages. In W.J. Savitch, E. Bach, W. Marsh, and G. Safran-Naveh, editors, *The Formal Complexity of Natural Language*, pages 320–334. D. Reidel, Dordrecht, 1987.
33. H.A. Simon. *The shape of automation for men and management*. Harper and Row, New York, 1965.
34. D. Stewart. *Interview with Herbert Simon*. Omni Magazine, June 1994.
35. L.G. Valiant. A theory of the learnable. *Communication of the ACM*, 27:1134–1142, 1984.



Logic Programming for Linguistics: A short introduction to Prolog, and Logic Grammars with Constraints as an easy way to Syntax and Semantics

Henning Christiansen

CBIT Institute, Roskilde University, Denmark
<http://www.ruc.dk/~henning/>, henning@ruc.dk

Introduction

This article gives a short introduction on how to get started with logic programming in Prolog that does not require any previous programming experience. The presentation is aimed at students of linguistics, but it does not go deeper into linguistics than any student who has some ideas of what a computer is, can follow the text. I cannot, of course, cover all aspects of logic programming in this text, and so we give references to other sources with more details.

Students of linguistics must have a very good motivation to spend time on programming, and I show here how logic programming can be used for modelling different linguistic phenomena. When modelling language in this way, as opposed to using only paper and pencil, your models go live: you can run and test your models and you can use them as automatic language analyzers. This way you will get a better understanding of the dynamics of languages, and you can check whether your model expresses what you intend.

Based on Prolog, I also introduce Definite Clause Grammars which is integrated in most Prolog systems: You can write a grammar in a straightforward notation, perhaps include different syntactic, semantic and pragmatic features – and with no additional effort, you can use it as an automatic language analyzer.

I show also another important extension to Prolog, called Constraint Handling Rules, which boosts these grammars with capabilities for capturing semantics and pragmatics by abductive reasoning, in a way that I claim is considerably simpler than mainstream formalisms; this part is to a large extent based on my own research.

Hardcore linguists may object that these approaches are too simplistic – and they are right (of course, they are always right ;-) – but this simplicity, I will reply, provides exposure to linguistic phenomena in a clarified and distilled form which is difficult to obtain by other means.

Finally I apologize for any errors, omissions, misspellings and mistakes, which I'm sure there are plenty of, as this article has been produced in a very short time. I am glad to receive any comments and questions.

All example programs can be downloaded from the following website: <http://www.ruc.dk/~henning/LP-for-Linguists>.



1 Prolog: Programming without programming

Prolog is one of the easiest programming languages to use for a beginner in programming: You only need to learn a few simple basic structures, and you can start on your own. Programs are given as plain text files which you can edit with any plain text editor.

1.1 Prolog, lesson 1: A program as a knowledge base of facts

The following is our first Prolog program; we will assume that it is kept in a file named `royal`.¹

```
% Danish Royal Family

parent( margrethe, frederik).
parent( margrethe, joachim).
parent( henrik, frederik).
parent( henrik, joachim).
parent( mary, christian).
parent( mary, isabella).
parent( frederik, christian).
parent( frederik, isabella).
parent( alexandra, nicolai).
parent( alexandra, felix).
parent( joachim, nicolai).
parent( joachim, felix).
parent( marie, little_henrik).
parent( joachim, little_henrik).
```

The very first character in the program “%” indicates that the rest of that line is a comment. The rest of this program consists of *facts*, in this example listing the parental relationships for a part of the Danish royal family. The meaning of the program is not that the program should be executed from beginning to end, one instruction at a time, but it is to be understood as a *knowledge base*.

¹ In some operation systems, it will best to give the file name an extension, e.g. `royal.txt`. The extension “.pl” is also common, but this makes many systems believe that the file is a Perl program, which is something completely different.



I suppose this meaning becomes clear to you, simply by looking at the program. You will also observe that these facts are written in a fixed format; here *parent* is called a *predicate*, and the names that appear, such as *margrethe*, are examples of *constant symbols* or, for short, *constants*.² It is important to notice that each fact must be ended with a period “.”.

We can run a program by asking *queries*. A query is a sort of question that the Prolog system tries to answer as good as it can. We will try some examples. The following shows a dialogue with a computer that has a version of Prolog installed; we assume that it is started by the command `prolog`, but this may vary; before you start, be sure to be in the directory that contains the program file `royal`. The following is a listing of the command window after a dialogue between a user and the Prolog system.

```
$ prolog
.....
| ?- [royal].
% compiling directory/royal...
yes
| ?- parent(margrethe, frederik).
yes
| ?- parent(margrethe, obama).
no
| ?- parent(margrethe, juan_carlos).
no
| ?- parent(margrethe, X).
X = frederik ? ;
X = joachim ? ;
no
| ?- parent(X,felix).
X = alexandra ? ;
X = joachim ? ;
no
| ?- halt.
$
```

² Some books and manuals also use the term, an *atom*, which is a bit misleading since an atom is something different in the mathematical logic on which Prolog is based.



You cannot see who (user or computer) wrote which part of the text, but I will explain. The \$ sign is the prompt from the computer's operating system, and the user starts Prolog by typing the command `prolog`. Where you see "`.....`", you will typically get a message saying which Prolog system and version you are using, but this is not interesting. The symbols "`| ?-`" are printed by the system, meaning that it expects input from the user. Our nice user first loads the program by writing "`[royal].`";³ this is the general syntax for loading in programs; notice the terminating period, and that is the same for any query that you type following "`| ?-`", and you should also type end-of-line at the end. Now the system gives a response, saying that it has accepted the program. which is now ready in its memory.

Now we can start asking queries, and assume that our nice user types "`parent(margrethe, frederik)`". When typed in like this, you should understand this as a question "*Is it true that ...*"; here the system answers "yes", which means that it has found out that, this is indeed true according to the program.⁴ In this case, it is easy for us to check that Prolog was right since the query matches a fact in the program. Let us try a more advanced query, "`parent(margrethe, juan_carlos)`". Here the system replies "no" meaning that the query cannot be shown to be true according to the program; we can easily check that this conclusion is correct.⁵

³ If your computer requires files with an extension as in "`royal.txt`" or "`royal.pl`", it often works to load the file without writing the extension, "`| ?-[royal]`". If that does not work, you may need to write the extension as well, e.g., "`| ?-[royal.txt]`". Notice when you do this that single quotes are essential, otherwise Prolog gets confused by the period and emits a weird error message,

⁴ You may notice the dubious usage of "true" and "truth"; what I mean here is not that something is true in the real world, but it is a logical consequence of the program. In fact, the computer has no coupling between the constant `margrethe` and a real living person, who happens to be Her Majesty, the Queen of Denmark. This meaning is reserved for humans, based on our intuition and knowledge about the world; if the program is wrong according to the real world, its answers will of course be wrong. The other way round, whether or not the program is correct, we can say that it defines a set of possible worlds, and something is stated to be true by Prolog only when it is true in all those possible worlds. However, this is too philosophical for us, so I shall leave it for now.

⁵ Notice that Prolog considers anything to be false, as indicated by "no", that cannot be shown to be correct by the program. This is also problematic as something might be true in the real world even if it is not mentioned in the program. In fact, it is difficult to imagine a program that contains all the knowledge about



Now our nice user tries something really advanced, namely to use a *variable* in the query “parent(margrethe, X).”; notice that variables are indicated by initial capital letter whereas a constant starts with a small letter. The meaning of such a query is a request for “which values of the variable makes my query true”. So when our nice user queries “parent(margrethe, X).”, it means that she wants to know which people that have margrethe as a parent. As it appears, the systems tells the nice user that there are two possibilities, namely X=frederik and X=joachim; also this time we can compare with the program text, that this is indeed a sound conclusion.

You should be aware that after each answer, when Prolog states a question mark “?” as shown, it should be understood as “do you want another solution?”. Here our nice user needs to type a semicolon “;” if she wants to confirm that she wants another solution; the final “no” should be taken as “no *more* answers”. If the first answer is sufficient, simply type end-of-line after the question mark.

The next query also uses a variable, but in a different position, namely for asking who are the parents of felix. It is important to learn from these example, that in Prolog, no specific positions of a predicate should be thought of as specific for “input” and others specific for “output”; you can use them as you please.

Finally, the query “halt.” is a command to Prolog that we want to stop and return to the operating system.

1.2 Prolog, lesson 2: Using variables to combine information and writing rules

As I wrote above, using a variable in a query was a suggestion for the system to fill in constants, so that the query becomes true. In fact, we can use several variables in a query and also, the same variable can appear several times. Let us consider an example, and assume that our nice user is interested to know, who the queen’s grandchildren are; obviously, this information is

the real world. So “false” in Prolog’s terms does not mean that it is really false, but rather that the program does not contain information about it. So it might be more correct to have the system state “I don’t know” rather than “no”, but that is too difficult to say; “no” is easier, and as soon as you know what a “no” means, this should not be a problem. — I promised in the last footnote not to include any more philosophical discussion, so this is definitely the very last footnote of this kind.



embedded in the program, but not in an explicit way. She now starts the system again, loads the program, and tries the following query.

```
| ?- parent(margrethe,X), parent(X,Y).
```

This is asking for pairs of values of *X* and *Y* which makes the query true. In other words, *X* should be a child of *margrethe*, and *Y* should be the child of the aforementioned child, i.e., *Y* should be a grandchild of *margrethe*. Let us see Prolog's answers when our nice user types semicolons after each to get more.

```
X = frederik,
Y = christian ? ;
X = frederik,
Y = isabella ? ;
X = joachim,
Y = nicolai ? ;
X = joachim,
Y = felix ? ;
X = joachim,
Y = little_henrik ? ;
no
```

It may be a bit difficult to read when you see everything at the same time, but each answer is ended by the semicolon typed by the user. So for example, *isabella* is a grandchild of *margrethe* because *isabella* has *frederik* as a parent, and *frederik* has *margrethe* as a parent.

Our nice user may become a bit tired both from writing the complicated expressions, each time having to get the *X*s and *Y*s right, and from seeing *margrethe*'s. There is a remedy in Prolog to this namely the possibility of defining a *rule* as part of the program. Our nice user suggests this rule:

```
grandparent(X,Z):- parent(X, Y), parent(Y, Z).
```

She adds it to the program file and reads in the program file once again to test it. She now types in the following query and a number of semicolons to get the following answers.

```
| ?- grandparent(margrethe,X).
X = christian ? ;
X = isabella ? ;
X = nicolai ? ;
```



```

X = felix ? ;
X = little_henrik ? ;
no

```

As it appears, this is exactly what she wants. The meaning of the rule is straightforward: in order to evaluate `grandparent(X,Z)` for which the program has no facts, evaluate instead `parent(X, Y)`, `parent(Y, Z)`, and return what was found for `X` and `Z`, thus saying nothing about the values of `Y` as it is completely local to the body of the clause.

Notice that we used the variable names freely as we liked. When using an `X` as the query, we do not have to consider whether the rule uses `X` for some other purpose; and if we have several rules in the program, their different uses of `X` do not get mixed up. So when, in the example above, we wrote `margrethe` in the query where the rule uses `X`, and we wrote also `X` in the query where the rule uses `Z`, they do not get mixed up. The system is clever enough to replace variables and values so everything works out the right way.

This finishes Prolog lesson 1 and 2, which is the core of Prolog and with which you can already write a lot of interesting programs.

1.3 Prolog, lesson 3: The rest of Prolog, with a focus on lists

Prolog includes a lot of other things, of which the most important are:

- a lot of standard built-in predicates so you do not need to write them yourself every time; any comprehensive Prolog textbook or manual will tell you about them,
- structures, so that we can use structural information in predicates, and not only constants such as `"margrethe"`; a special kind of structures is lists that I will show below as they are important for language processing,
- some devices which makes it possible for you to affect the way Prolog is searching in its knowledge base for rules and facts in order to answer the queries; this can give essential speed-up to large programs but I ignore all that in this article.

The following is an example of a Prolog list.

```
[once, upon, a, time]
```



It includes four constants, and as you see, I anticipate the use of lists to represent text. Prolog gives us some notation to work with lists, as we show in the following program.

```
first(H, [H | _]).
rest(R, [_ | R]).
```

Firstly, the underline character is used as a so-called anonymous variable; it adds nothing conceptually new to what we have already seen and is not restricted to lists. It can be used for a variable that we only use once in a rule (so there is no reason to give it an explicit name (think!)), which means that we do not care about its actual value. The vertical bar inside the list brackets is a special notation for lists, and separates the first element from the list of remaining elements. So for example, when `[once, upon, a, time]` is matched with `[A|B]`, it will lead to `A=once` and `B=[upon, a, time]`. Here are some queries to the program above together with its answers; the program is in a file called `firstlast`.

```
$ prolog
| ?- [firstlast].
% compiling directory/firstlast...
yes
| ?- first(F, [once, upon, a, time]).
F = once ?
yes
| ?- rest(R, [once, upon, a, time]).
R = [upon,a,time] ?
yes
| ?- halt.
$
```

I will not spend more time on this example, but instead show a linguistically inspired example. I will later introduce a grammar notation which makes it easier to write, but now our point is to illustrate the use of lists for language. The following program (file `sheeps`) uses Prolog and its list notation to define a grammar for sheeps' utterances.

```
sheeptalk([]).
sheeptalk([M|Ms]):- sheepsound(M), sheeptalk(Ms).
sheepsound(mah).
sheepsound(meeeeeh).
```



The first rule tells that sheep can keep quiet when necessary, `[]` is a notation for the empty list. The second rule explains in a recursive way that `sheeptalk` consists of `sheepsounds`. In practice this means that it will clip off one atom at a time and check if it is a `sheepsound`, i.e., `mah` or `meeeeeh`. The following shows it at work.

```
$ prolog
.....
| ?- [sheeps].
% compiling directory/sheeps...
yes
| ?- sheeptalk([mah,mah,meeeeeh]).
yes
| ?- sheeptalk([mah,mah,mouuuuuuuuuuuuuuh]).
no
| ?- halt.
$
```

This is the essence of language analysis in Prolog; notice that you can think of such a program as a grammar, and that Prolog can automatically use it as an analyzer. And with a bit of imagination, you may be able to see that we can extend this with different predicates for nouns, verbs, adjectives, etc., and that an elaborate set of rules can express how some natural sentences may look like. However, in the next section, I will introduce a special grammar notation that most Prolog systems can use.

1.4 More reading

There are several good books that introduce to and go in depth with Prolog; for computer science students, I have good experience of using Bratko's book [3], but the first half of the book is also fairly accessible to other people. The online, and now also paperback, book [2] may be easier to access for linguists. I will also refer to my own course notes [10] which are biased towards applications in artificial intelligence, including computational linguistics, and databases; if you skip the very few mathematical formulas that appear occasionally, it can give you an easily read (hmmm, well, fairly easily read) introduction to these areas. The notes have the advantage that they also introduce Constraint Handling Rules, which we apply to semantic-pragmatic analysis below.



Prolog was originally developed by a research group in Marseilles led by Alain Colmerauer in the 1970s, and spreading of it was strongly promoted by D.H.D. Warren's first efficient implementation of Prolog [36] and R.A. Kowalski's book from 1979 [29]; since 1982, there have been annual conferences, ICLP, International Conference on Logic Programming.

2 Definite Clause Grammars

Now you have understood the basic mechanics of Prolog, I will introduce you to its grammar notation, called *Definite Clause Grammars* or *DCGs* among friends, by means of an example. You can write such rules directly in your Prolog program files, and you can mix Prolog and DCGs whenever you wish.

2.1 DCG, lesson 1: The basic grammar notation and syntax analysis

The sheeps program shown above can be written alternatively using grammar rules as follows; we assume it is contained in a file `sheepsGrammar`.

```
sheeptalk--> [].
sheeptalk--> sheepsound, sheeptalk.
sheepsound--> [mah].
sheepsound--> [meeeeeh].
```

You can see that we avoid explicitly clipping off constants one at a time, and we do not have to write list arguments explicitly. In a grammar, we may use *nonterminal (symbol)s* such as `sheeptalk`, and *terminal symbols* that are written in square brackets (i.e., the list notation re-used). In fact Prolog will translate, behind you back, a grammar such as `sheepsGrammar` into a Prolog program that resembles the sheeps Prolog program that I showed above. To query a grammar, i.e., to use it to analyse text, we need to use a special built-in predicate called `phrase`. It is shown at work below:

```
$ prolog
.....
| ?- [sheepsGrammar].
% compiling directory/sheepsGrammar...
| ?- phrase(sheeptalk,[mah,mah,meeeeeh]).
yes
```



```
| ?- phrase(sheeptalk,[mah,mah,mouuuuuuuuuuuuuuh]).
no
| ?- halt.
$
```

As you can see, DCGs provide a formal grammar notation, and you can use the Prolog system to tests examples to convince yourself that the grammar actually expresses what you have in mind. I claim that this is a very good reason for students of linguistics to work with these tools.

2.2 DCG, lesson 2: Adding features

A grammar can do more than just say yes and no, because we can add all kinds of features to the nonterminals, in a very similar way to how we used arguments for the predicates.

I will use a simplistic extension to the `sheepsGrammar` to illustrate this. I will consider how much grass a sheep needs to eat in order to perform a given speech; let us assume that a sheep needs one lump of grass to say mah and three lumps to say meeeeh. For each syntactic phrase, we attach a feature that counts the total number of lumps for that phrase. This can be expressed as follows; you should notice the following details: the curly brackets `{...}` inside a grammar rule indicate a piece of Prolog code that should be interpreted whenever the given rule applies, and secondly Prolog's strange way of doing arithmetic by the "is" construction used below in order to perform an addition. Let the file `sheepsGrammarGrass` contain the following grammar.

```
sheeptalk(0)-->[].
sheeptalk(C)--> sheepsound(C1), sheeptalk(C2), {C is C1+C2}.
sheepsound(1)--> [mah].
sheepsound(3)--> [meeeeh].
```

It works as follows.

```
$ prolog
.....
| ?- [sheepsGrammarGrass].
% compiling directory/sheepsGrammar...
| ?- phrase(sheeptalk(C),[mah,mah,meeeeh]).
C = 5 ?
```



```
yes
| ?- halt.
$
```

Finally I show a more interesting grammar for a subset of English; here I add a feature for number whenever it is relevant, and express the constraint that the number for noun phrase must match the number for the following verb phrase. Notice that number (indicated by variables called “N”) can assume the values *plus* and *sing*. We write the grammar in the text file called *english* as follows.

```
s --> np(N), v(N), np(_).
np(N) --> noun(N).
np(plur) --> noun(_), [and], np(_).
noun(sing) --> [joachim].
noun(sing) --> [alexandra].
noun(sing) --> [marie].
noun(plur) --> [dogs].
v(sing) --> [likes].
v(plur) --> [like].
```

The following queries show it at work; I suggest that you inspect each query in detail and understand exactly why it answers as it does.

```
$ prolog
.....
| ?- [english].
% compiling directory/english...
| ?- phrase(s, [joachim,likes,dogs]).
yes
| ?- phrase(s, [joachim,like,dogs]).
no
| ?- phrase(s, [marie,and,alexandra,likes,joachim]).
no
| ?- phrase(s, [marie,and,alexandra,like,joachim]).
yes
| ?- halt.
$
```

You can also extend your grammar with structures that represent syntax trees, so that when you analyze a sentence, you get as a result the tree that



represents the phrase structure of that sentence. It is straightforward to do so, and you can do it yourself, provided that you find a textbook or good course notes and read about structures in Prolog.

2.3 More reading

Definite clause grammars (DCG) were first presented in 1975 by A. Colmerauer [18] under the name of *grammaires de métamorphose*, and they got their final shape and name as DCGs in 1980 [31]. Any good Prolog textbook will have a section on Definite Clause Grammars, and they are included in virtually all available Prolog systems.

3 A brief introduction to Constraint Handling Rules, CHR, and their application for abductive reasoning

The term abduction usually refers to a kind of criminal act, quite different from the specific meaning that what I use it for here, and abductive reasoning sounds weird to most people.

I first give an introduction to the topic taken from [10], and then I introduce the language of Constraint Handling Rules by means of a few examples of how they can be used for adding abductive reasoning to Prolog. Then, I combine this with the grammar notation introduced above.

You may find the name and term “constraints” a bit confusing; this is a consequence of the application that CHR was originally designed for, which we discuss briefly in section 3.4 below.

Most applications of abductions, including the methods I introduce below, are used for diagnosis and planning; I will not go into such examples here, but you may try to think about the similarities between language interpretation and diagnosis.

3.1 Deduction, abduction, and induction in logic programming

The philosopher C.S. Peirce (1839–1914) is considered a pioneer in the understanding of human reasoning, especially in the specific context of scientific discovery. His work is often cited in computer science literature but probably only a few computer scientists have read Peirce’s original work. I recommend [21] as an overview of Peirce’s influence seen from the perspective of computer science.



Peirce postulated three principles as *the* fundamental ones:

- **Deduction**, reasoning within the knowledge we have already, i.e., from those facts we know and those rules and regularities of the world that we are familiar with. E.g., reasoning from causes to effects:
"If you make a fire in the living room, you will burn down the house."
- **Induction**, finding general rules from the regularities that we have experienced in the facts that we know; these rules can be used later for prediction:
"Every time I made a fire in my living room, the house burnt down, aha, ... the next time I make a fire in my living room, the house will burn down too".
- **Abduction**, reasoning from observed results to the basic facts from which they follow. Quite often it means from an observed effect to produce a qualified guess for a possible cause:
"The house burnt down. Perhaps my cousin made a fire in the living room again."

In fact, Peirce had alternative theories and definitions of abduction and induction; I have adopted the so-called syllogistic version, cf. [21]. I can replicate the three in logic programming terms:

- A Prolog system is a purely deductive engine. It takes a program of rules and facts, and it can calculate or check the logical consequences of that program.
- Induction is difficult; methods for so-called inductive logic programming (ILP) have been developed, and by means of a lot of statistics and other complicated machinery, they synthesize rules from collections of "facts" and "observations". I can refer to [4]⁶ for an overview of different applications. Inductive logic programming has been successfully applied for molecular biology concerned with protein molecule shapes and human genealogy. See [30] for an in-depth treatment of ILP methods.
- Abductive logic programming; roughly means from a claim of goal that is required to be true (i.e., being a consequence of the program), to extend to program with facts so that the goal becomes true. See [27] for an overview. Abduction has many applications; I may mention planning (e.g., the goal is "successful project ended" and the facts to be derived are the detailed steps of a plan to achieve that goal), diagnosis (goal

⁶ A bit old; if you are interested, you should search for more recent overview papers and consult proceedings of the recent ILP conferences; see <http://www.informatik.uni-trier.de/~ley/db/conf/ilp/index.html>.



is observed symptoms, the facts to be derived comprise the diagnosis, i.e., which specific components of the organism or technical system that malfunction). An important area for abduction is language processing, especially discourse analysis (the discourse represents the observations, the facts to be derived constitute an interpretation of that discourse). We will look into some of these in more detail below and give references.

However, we should be aware that while deduction is a logically sound way of reasoning, this is generally not the case for abduction and induction. Let me make a simple analysis for abduction. Assume a logical knowledge base $\{a \rightarrow c, b \rightarrow c\}$ where the arrow means logical implication. If we know c , an abductive argument may propose that a is the case. However, this is not necessarily true as it might that b is the case and not a . Or it could even be the case that none of a and b are the case, and that there is another and unknown explanation for c . Abduction is often described as reasoning to the best explanation. i.e., best with respect to the knowledge we have available.

3.2 Introducing Constraint Handling Rules by examples of abduction

Constraint Handling Rules [22], CHR, is a declarative, rule-based language for writing constraint solvers and is now included as an extension of several versions of Prolog. Operationally and implementation-wise, CHR extends Prolog with a constraint store, and the rules of a CHR program serve as rewriting rules over constraint stores. CHR is declarative in the sense that its rules can be understood as logical formulas. I show first a program in Prolog that does not use CHR and we analyze its deficiencies; it is given as the file `happy1`.

```
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

It is supposed to describe how someone can become happy, which, however, does not fit exactly with Prolog's mode of working, as we will see. We ask now the following query with the intension of finding out how someone with the name `henning` can be happy.

```
| ?- happy(henning).
! Existence error in user:rich/1
! procedure user:rich/1 does not exist
! goal: user:rich(henning)
```



It goes wrong because Prolog needs to investigate calls to `rich`, which is not defined by any facts or rules. By giving a suitable command to Prolog (which I don't show here), we can get rid of the error message, so that a call to a predicate with zero facts and rules always fails (as opposed to crashing), which is more in accordance with a logical meaning of the Prolog program.⁷ In this case we would get the answer `no` instead since the predicates `rich`, `professor` and `has` are all empty, but this is still not satisfactory according to our intension with the query.

What we wanted to achieve, was one or more explanations of how we could get the conclusion `happy(henning)`, and to do this, we must make a part of the program dynamic in the sense that the system should be able to add facts to see if that made the goal succeed. Now you may see the relationship with abductive reasoning, which, as I have shown, is beyond plain Prolog's capabilities.

We can now use CHR to declare the predicates `rich`, `professor` and `has` as *constraints*, in the sense that they are now governed by the CHR system. We do this in the next version of the program, `happy2`; the first line is necessary for making CHR available.

```
:- use_module(library(chr)).
:- chr_constraint rich/1, prof/1, has/2.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Having these predicates declared as constraints will have the effect that they 1) are not unknown anymore, and 2) whenever they are called, the system will add the calls to its constraint store. At the end, the collected constraint store is printed out as part of the answer. We test the `happy2` program and get the following results.⁸

⁷ There is a very good reason, though, why Prolog as default emits an error message rather than silently failing if a non-existing predicate is called. Can you imagine, if you have a very big program over several thousand lines, and you have misspelled one occurrence of a predicate; the error message will help you to locate the error while a failure would make it almost impossible to detect if, e.g., the program simply answers “no”.

⁸ **Important note concerning SWI Prolog:** Some older versions do not print out the constraint store when the program finishes; if you experience this problem, check the manual for the version you are using, to find the command that makes it print the constraint store. Otherwise, there is not much fun in using CHR for abduction!



```
| ?- happy(henning).
rich(henning) ? ;
professor(henning),
has(henning,nice_students) ? ;
no
```

As you can see, the two alternative answers say that there are two ways that `happy(henning)` can be true, namely if either the constraint store contains `rich(henning)` or, alternatively, `professor(henning)` and `has(henning, nice_students)`.

I will relate these answers to abductive reasoning as follows:

If we forget everything about CHR and type in, say, `rich(henning)` as a part of the program, then `happy(henning)` will succeed, i.e., answer “yes”.

However, we can improve this program even further and make it better to reflect the real world. It is a fact that university professors are much lower paid than people in the industry with less education, and we always complain about this. We should somehow express this in our program, and here the rules of CHR come in handy. Rules in CHR operate on the constraint store, and a rule *fires*, whenever the total set of constraints in the store makes it possible for that rule to apply. We show this in an improved version of the program, `happy3`.

```
:- use_module(library(chr)).
:- chr_constraint rich/1, prof/1, has/2.
prof(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

The construction written with “`==>`” is a CHR rule of the kind called a *propagation rule*. The meaning is that when its head (the left hand side) matches constraints in the store, the body (right hand side) is executed; in the example the body amounts to “`fail`” which will cause the system to try another branch. Now let us see how this program works; notice that the program does not know that `professor(henning)`, so we need to state this as part of the query to get the right answers.

```
| ?- happy(henning), professor(henning).
professor(henning),
professor(henning),
has(henning,nice_students) ? ;
```



no

Here we get only one answer, namely that `professor(henning)` and `has(henning,nice_students)`. The alternative postulation a professor to be rich is removed due to the CHR rule. You may notice that the constraint `professor(henning)` is repeated in the answer; this is due to some technical reasons that I will not spend time on explaining, it does not mean any thing.

This last example basically shows the part of CHR that you need to know how to use it for abductive language interpretation, as shown below.

Finally, I will comment on some terminological confusion that appears because this way of doing, involves usages from different areas.

- “*Constraint*” refers in CHR context to predicates that have been declared in as such, and that are treated by the system in a special way. We use CHR constraints here for what in the tradition of abductive reasoning is called *abducibles*.
- “*Integrity constraint*” refers in database theory and in abductive reasoning not to the simple piece of information, but to general knowledge about the world, about what is possible and what is not. Above, we used a CHR rule to describe an integrity constraint.
- Finally, as you have noticed, Prolog rules and CHR rules are something completely different, so referring to “*a rule*” may be ambiguous.

3.3 Details of CHR

Most of my readers may skip this subsection as you can make interesting linguistic applications, by generalizing from the examples above. The rest of this section is taken verbatim from [10], and may contain a few terms that you may be unfamiliar with.

CHR takes over the basic syntactic and semantic notions from Prolog and extends them with its specific kinds of rules. The execution of CHR programs is based on a *constraint store*, and the effect of applying a rule is to change the effect of the store. For a program written in a combination of Prolog and CHR, the system switches between two tow. When a Prolog goal is called, it is executed in the usual top-down (or goal-directed) way, and when a Prolog rule calls a CHR constraint, this will be added to the constraint store — then the CHR rules apply as far as possible, and control then returns to the next Prolog goal.



Technically speaking, CHR constraints are first-order atoms whose predicates are designated constraint predicates, and a constraint store is a set of such constraints, possible including variables that are understood existentially quantified at the outermost level. A constraint solver is defined in terms of rules which can be of the following two kinds.

Simplification rules: $c_1, \dots, c_n \iff Guard \mid c_{n+1}, \dots, c_m$

Propagation rules: $c_1, \dots, c_n \implies Guard \mid c_{n+1}, \dots, c_m$

The c 's are atoms that represent constraints, possibly with variables, and a simplification rule works by replacing in the constraint store, a possible set of constraints that matches the pattern given by the *head* c_1, \dots, c_n by the constraints given by the *body* c_{n+1}, \dots, c_m , although only if the condition given by *Guard* holds. A propagation rule executes in a similar way but without removing the head constraints from the store. What is to the left of the arrow symbols is called the *head*⁹ and what is to the right of the guard the *body*. The declarative semantics is hinted by the applied arrow symbols (bi-implication, resp., implication formulas, with variables assumed to be universally quantified) and it can be shown that the indicated procedural semantics agrees with this. This is CHR explained in a nutshell.

CHR provides a third kind of rules, called *simplagation rules*, which can be thought of as a combination of the two or, alternatively, as an abbreviation for a specific form of simplification rules.

Simplagation rules: $c_1, \dots, c_i \setminus c_{i+1}, \dots, c_n \iff Guard \mid c_{n+1}, \dots, c_m$

which can be thought of as: $c_1, \dots, c_n \iff Guard \mid c_1, \dots, c_i, c_{n+1}, \dots, c_m$

In other words, when applied, c_1, \dots, c_i stays in the constraint store and c_{i+1}, \dots, c_n are removed.

In practice, the body of CHR rules can include any executable Prolog expression including various control structures and calls to Prolog predicates. Similarly, Prolog rules and queries can make calls to constraints which, then, may activate the CHR rules.

The guards can be any combination of predicates (built-in or defined by the programmer) that test the variables in the head, but in general guards should not change the values of these variables or call other constraints; in these cases, the semantics gets complicated, see references given above if

⁹ Some authors call each constraint to the left of the arrow a head, and with that terminology, CHR has multi-headed rules.



you are interested in the details. Finally, guards can be left out together with the vertical bar, corresponding to a guard that always evaluates to true.

The following example of a CHR program is adapted from the reference manual [33]; from a knowledge representation point of view it may seem a bit strange, but it shows the main ideas. It defines a little constraint solver for a single constraint `leq` with the intuitive meaning of less-than-or-equal. The predicate is declared to be an infix operator to enhance reading, but this is not necessary (`X leq Y` could be written equivalently as `leq(X,Y)`).

```
:- use_module(library(chr)).
handler leq_handler.
constraints leq/2.
:- op(500, xfx, leq).

X leq Y <=> X=Y | true.
X leq Y , Y leq X <=> X=Y.
X leq Y \ X leq Y <=> true.
X leq Y , Y leq Z ==> X leq Z.
```

The first line loads the CHR library which makes the syntax and facilities used in the file available. The `handler` directive is not very interesting but is required. Next, the constraint predicates are declared as such (here only one such predicate) and this informs the Prolog system that occurrences of these predicates should be treated in a special way.

The program consists of four rules, one propagation, two simplifications, and one simplagation. The first simplification describes the transitivity of the `leq` constraints. If, for example, the constraints `a leq b` and `b leq c` are called, this rule can fire and will produce a new constraint `a leq c` (which in turn may activate other rules).

The second rule is a simplification rule which will remove the two constraints and unify the arguments. Intuitively, the rule says that if some `X` is less than or equal to some `Y` and the reverse also holds, then they should be considered equal (antisymmetry). With constraint store $\{a \text{ leq } Z, Z \text{ leq } a\}$, the rule can apply, by removing the two constraints and unifying variable `Z` with the constant symbols `a`.

Consider a slightly different example, the constraint store $\{a \text{ leq } b, b \text{ leq } a\}$. Again, the rule can apply, by removing the two constraints from the store and calling `a=b`. This will fail as `a` and `b` are two different constant symbols.



Notice that CHR is a so-called *committed choice* language in the sense that when a rule has been called, a failure as exemplified above will not result in backtracking. I.e., in the example, the observed failure will **not** add $\{a \text{ leq } b, b \text{ leq } a\}$ back to the constraint store so other and perhaps more successful rules may be tried out. However, when CHR is combined with Prolog, a failure such as the one shown will cause Prolog to backtrack, i.e., it will undo the addition of the last of the two, say $b \text{ leq } a$, and go back to the most recent choice point.

The simplification rule $X \text{ leq } Y \text{ } \Leftarrow \text{ } X=Y \mid \text{true}$ will remove any leq constraint from the store with two identical arguments. This illustrates a fundamental difference between Prolog and CHR. Where Prolog uses unification when one of its rules is applied to some goal, CHR uses so-called matching. This means that the mentioned rule will apply to $a \text{ leq } a$ but not to $a \text{ leq } Z$. In contrast, the application of Prolog rule $p(X,X) : - \dots$ to $p(a,Z)$ will result in $a=Z$ before the body is entered.

The third rule in the program above is a simplagation rule $X \text{ leq } Y \setminus X \text{ leq } Y \text{ } \Leftarrow \text{ } \text{true}$ which serves the purpose of removing duplicate constraints from the store.

We consider the following query and see how the constraint store changes.

?- $C \text{ leq } A, B \text{ leq } C, A \text{ leq } B.$

Calling the first constraint triggers no rule and we get the constraint store $\{C \text{ leq } A\}$. Calling the next one will trigger the transitivity rule (the last rule), and we get $\{C \text{ leq } A, B \text{ leq } C, B \text{ leq } A\}$. The last call in the query will trigger a sequence of events. When $A \text{ leq } B$ is added to the constraint store, it reacts, so to speak, with $B \text{ leq } A$ and the second rule applies, removing the two but resulting in the unification of A and B ; for the sake of clarity, let us call the common variable, which is referred to by both A and B , $V1$. Now the constraint store is $\{C \text{ leq } V1, V1 \text{ leq } C\}$. The same rule can apply once again, unifying C and $V1$, so that the result returned for the query is the empty constraint store and the bindings $A=B=C$.

In general, when a query is given to a CHR program (or a program written in the combined language of CHR plus Prolog), the system will print out the final constraint store together with Prolog's normal answer substitution. An alternative solution can be asked for as in traditional Prolog by typing a semicolon.



I end the presentation of CHR by showing a few simple examples taken from the CHR web site [5]. This program by Thom Frühwirth evaluates the greatest common divisor of positive numbers.

```
:- use_module( library(chr)).
handler gcd.
constraints gcd/1.

gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N=<M | L is M-N, gcd(L).
```

Here are a few test queries.

```
?- gcd(2),gcd(3).
?- X is 37*11*11*7*3, Y is 11*7*5*3, Z is 37*11*5, gcd(X),
   gcd(Y), gcd(Z).
```

The following program generates the prime numbers between 1 and n when given the query `?- primes(n)`. It was written by Thom Frühwirth and adapted by Christian Holzbaur.

```
:- use_module(library(chr)).
handler primes.
constraints primes/1, prime/1.

primes(1) <=> true.
primes(N) <=> N>1 | M is N-1, prime(N), primes(M).
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

3.4 More reading

Constraint Handling Rules (CHR) were developed by T. Frühwirth from around 1992, first publication [24], in order to have a declarative language for writing constraint solvers for, e.g., working with arithmetic in logic programming. Later, it turned out that CHR was suited to a much wider class of applications as illustrated in the present article. The use of CHR for abductive reasoning was discovered by S. Abdennadher and myself in 2000 [1] and later the ideas have been refined in my own work, largely in an inspiring collaboration with Veronica Dahl, see, e.g., [6, 11–15].



A recent book [23] gives a thorough, mainly theoretical treatment of all aspects of CHR, and the collection [32] gives an overview of recent applications and developments concerning CHR. See also [22,25] for good overview papers and the CHR website

<http://www.cs.kuleuven.be/~dtai/projects/CHR>.

Since 2004, there have been annual workshops on CHR.

4 Language interpretation as abduction in Prolog+CHR

Now we have all the tools for doing abductive language interpretation: We have the DCG grammar notation for the syntax, and I will show how CHR can take care of a large portion of the semantic-pragmatic analysis. In fact, it is interesting to see how the use of abduction tends to remove the borderline between semantics and pragmatics.

4.1 Introducing abductive interpretation by examples

The following example was developed when I gave a talk for students at GRLMC in Tarragona, so that sets the context for the example. It may be possible that some people attend the talk while others are away; furthermore, we will be interested in who is able to see whom. Note that the example is not always perfect from an intuitive point of view, but it shows the method.

We make a first suggestion for a grammar that uses CHR to extract (a selected part of) the meaning of a given discourse. In this version, in file `discourse1`, we do not include any CHR rules. Notice that instead of having a general rule for sentences, we have specialized rules for the different sort of sentences that we want to analyze. This is not essential, but made in order to simplify this example; for larger applications, it may be advisable to use a more homogeneous format.

```
:- use_module(library(chr)).
:- chr_constraint at/2, sees/2.
story --> [] ; s, ['.'], story.
s --> np(X), [sees], np(Y), {sees(X,Y)}.
s --> np(X), [is,at], np(E), {at(E,X)}.
s --> np(X), [is,on,vacation], {at(vacation,X)}.
np(pedro) --> [pedro].
```



```

np(maria) --> [maria].
np(loli)   --> [loli].
np(grlmc) --> [grlmc].
np(hennings_talk) --> [hennings,talk].
np(vacation) --> [vacation].

```

Let us show a few examples of using this grammar for language analysis. The following query analyzes a very simple sentence and represents its meaning as a CHR constraint.

```

| ?- phrase(story, [pedro,is,at,grlmc,','.']).
at(grlmc,pedro) ? ;
no

```

Let us try another, longer discourse:

```

phrase(story, [pedro,sees,maria,','.', pedro,sees,loli,','.',
               pedro,is,at,grlmc,','.', maria,is,at,hennings,
               talk,','.',loli,is,on,vacation,','.']]).
at(vacation,loli),
at(hennings_talk,maria),
at(grlmc,pedro),
sees(pedro,loli),
sees(pedro,maria) ? ;
no

```

It appears that each sentence is “translated” into a formal form, but there is not much semantic-pragmatic processing involved. So let us add a few CHR rules to express a bit of simple every-day knowledge. The first rule says that if someone is at GRLMC, he or she is also in Tarragona; the next one says that anyone can only be in one location (it uses a so-called simpagation rule which removes that last of the matched in constraints, so that we avoid duplicate constraints). Next we express that if someone is at my talk, he or she is also at GRLMC, and finally, if someone is on vacation, he or she is not in Tarragona.¹⁰

¹⁰ The `diff` constraint is a device that ensures that two items need to be different for the rest of the discourse. You may use instead Prolog’s built-in `dif` (one `f`) instead, but my handcrafted version gives more readable output. It can be defined as follows; you do not need to read this; I include it for completeness only and to indicate that I have not hidden any code under the carpet to get it to work.



```

at(grlmc,X) ==> in(tarragona,X).
in(Loc1,X) \ in(Loc2,X) <=> Loc1=Loc2.
at(hennings_talk,X) ==> at(grlmc,X).
at(vacation,X) ==> in(Loc,X), diff(Loc,tarragona).

```

During an analysis of the discourse, these rules will fire as soon as they can and do some simple reasoning on the constraint store as the analysis proceeds. The program `discourse2` also contains these rules.

```

| ?- phrase(story, [pedro,sees,maria,',' , pedro,sees,loli,
                    ',' ,pedro,is,at,grlmc,',' , maria,is,at,
                    hennings,talk, ',' ,loli,is,on,vacation,',' ]).
at(vacation,loli),
at(grlmc,maria),
at(hennings_talk,maria),
at(grlmc,pedro),
in(_A,loli),
in(tarragona,maria),
in(tarragona,pedro),
sees(pedro,loli),
sees(pedro,maria),
diff(_A,tarragona) ? ;
no

```

As it appears, the meaning extracted from the discourse now also includes for each person, in which place he or she is. Note that `loli` is in some place, referred to be a variable written by the system as “`_A`”; we do not know where this place is, except that it is not `tarragona`.

We will now make one last extension, `discourse3`, of the program to indicate who can see whom. If you are able to see someone then you are both in the same place, e.g., `Tarragona`, *or* you contact the person using `skype`. The following CHR rule needs a bit of explanation. The semicolon in the body signifies a logical “or” so that the system will try out both possibilities if asked for more answer or if the first alternative leads to a

```

:- chr_constraint diff/2.
diff(X,X) <=> fail.
diff(A,B) \ diff(A,B) <=> true.
diff(A,B) \ diff(B,A) <=> true.
diff(A,B) <=> ?=(A,B) | true.

```



failure. Secondly, you should ignore the irrelevant “true |” in the rule: there is a design bug in the CHR syntax so that when you use a semicolon in the body, you need to write it like this; there is no reason to make any effort to understand why.

```
see(X,Y) ==> true | (in(L,X), in(L,Y) ;
in(Lx,X), in(Ly,Y), diff(Lx,Ly), skypes(X,Y)).
```

Let us try the usual query again with the program that contains all the rules shown so far.

```
| ?- phrase(story, [pedro,sees,maria,',' , pedro,sees,loli,
                    ',' ,pedro,is,at,grlmc,',' , maria,is,at,
                    hennings,talk, ',' ,loli,is,on,vacation,',' ,]).
at(vacation,loli),
at(grlmc,maria),
at(hennings_talk,maria),
at(grlmc,pedro),
in(_A,loli),
in(tarragona,maria),
in(tarragona,pedro),
see(pedro,loli),
see(pedro,maria),
skypes(pedro,loli),
diff(tarragona,_A) ? ;
no
```

We notice that the only answer is one in which Pedro sees Loli via skype, since the other option that they are in the same place is not possible: Pedro is in Tarragona and Loli is somewhere which is not Tarragona. This example has illustrated how the CHR rules can process the bits of information generated for each sentence and form it into a knowledge base, that also contains knowledge that is not expressed directly in the discourse, but is somehow necessary for the discourse to be made.

There are still a few imperfections in this grammar, for example that the *sees* relationship is not symmetric, but we would expect that if *A* sees *B* then *B* also sees *A*; this is easy to repair (when you are familiar with CHR), but there is no reason to spend more time on this here.



4.2 More reading

The principle of seeing language interpretation as abduction was first formulated in [26], which is a highly referenced paper from 1993. Abduction in logic programming was studied from around 1990 or before with [28] as a central reference; see the following overview papers [20,27]. The use of abduction implemented with CHR starts around 2000 with my own work; the first references are [6,7]. Later I developed these ideas together with Veronica Dahl, which led to the combined use of DCG and CHR as demonstrated above. The work with Veronica also resulted in the Hyprolog system which is briefly described next.

In [16,17], we have developed a realistic example of a CHR based grammar, which reads so-called use cases and produces UML diagrams. Use cases are used for the sort of system analysis that is made for the development of complex computerized systems that are typically used in large and complex organizations; use cases are small stories about what goes in the organization. A UML diagram, on the other hand, is a graphical representation of which classes of objects appear and their mutual relationships.

5 One step further: Hyprolog

Hyprolog is a system thought out by Veronica Dahl and myself, which puts an additional set of facilities on top of Prolog+DCG+CHR. The syntax for declaring abducible predicates is different (in Hyprolog, we call them *abducibles* rather than *chr_constraints*), and a few more aspects of abduction not described here are supported. Most notably, Hyprolog includes so-called *assumptions* that work very much like *abducibles*, but they also reflect the time which is implicit in a discourse — some things are said before and after certain other things — and they have explicit creation and applications.

I will not explain Hyprolog in detail, but you can refer to the articles [13,14] and the Hyprolog User's Guide which is available at [9] together with source code and examples. First we describe these new devices, assumptions, and then we sketch a larger example available from [9].

5.1 Assumptions: Like abduction but with time

The text in this subsection is taken from [14], written together with Veronica Dahl.



Assumptive logic programs [19] are logic programs augmented with a) linear, intuitionistic and timeless implications scoped over the current continuation, and b) implicit multiple accumulators, particularly useful to make the input and output strings invisible when a program describes a grammar (in which case we talk of Assumption Grammars [19]). More precisely, we use the kind of linear implications called *affine* implications, in which assumptions can be consumed at most once, rather than exactly once as in linear logic. Although intuitively easy to grasp and to use, the formal semantics of assumptions is relatively complicated, basically proof theoretic and based on linear logic [19, 34, 35]. Here we use a more recent and homogeneous syntax for assumptions introduced in [8]; we do not consider accumulators, and we note that Assumption Grammars can be obtained by applying the operators below within a DCG.

$+h(a)$	Assert linear assumption for subsequent proof steps. Linear means “can be used once”.
$*h(a)$	Assert intuitionistic assumption for subsequent proof steps. Intuitionistic means “can be used any number of times”.
$-h(X)$	Expectation: consume/apply existing int. assumption.
$=+h(a), =*h(X), =-h(X)$	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

A sequential expectation cannot be met by timeless assumption and vice versa, even when they have the same name. In [35], a query cannot succeed with a state which contains an unsatisfied expectation; for simplicity (and to comply with our implementation), this is not enforced in HYPROLOG but can be tested explicitly using a primitive called `expectations_satisfied`. Assumption grammars have been used for natural language problems such as free word order, anaphora, coordination, and for knowledge based systems and internet applications.

5.2 Sketch of an example Hyprolog program

A grammar for a subset of English is available at the Hyprolog website [9], click “Sample Hyprolog programs” and then “shootingLucky-LukeAdvanced”.¹¹ It includes pronoun resolution, in which we only allow

¹¹ There are some mistakes in the version at the Hyprolog website. A corrected one is available at <http://www.ruc.dk/~henning/LP-for-Linguists>.



backward references, so that “he” can only refer to a male character which has already been mentioned, and “they” can only refer to a group of at least two people already mentioned. The sentence in question refers to a world where people are shooting at each other, and those who have been shot, cannot shoot after that event. Each event is time-stamped according to the sentence in which it appears.¹² Here is an example of a query and its answer; as we see there is only one possible answer.

```
| ?- phrase(discourse, [luckyLuke,shoots,jackDalton,
                        calamityJane,shoots,averellDalton,
                        they,shoot,joeDalton])).
event(2,shooting,[calamityJane,luckyLuke],joeDalton),
event(1,shooting,calamityJane,averellDalton),
event(0,shooting,luckyLuke,jackDalton),
dead(2,joeDalton),
dead(1,averellDalton),
dead(0,jackDalton),
alive(2,luckyLuke),
alive(2,calamityJane),
alive(1,calamityJane),
alive(0,luckyLuke),
'*acting'(masc,joeDalton),
'*acting'(masc,averellDalton),
'*acting'(fem,calamityJane),
'*acting'(masc,jackDalton),
'*acting'(masc,luckyLuke) ? ;
no
| ?-
```

For example, you can see that “they” in the last sentence refers to Calamity Jane and Lucky Luke as they are the only persons mentioned still alive at the time for the shooting. Assumptions such as `'*acting'(masc,jackDalton)` are used for pronoun resolution that also involves the semantic reasoning that only live people can shoot.

As a final example, we illustrate how we can add a context to a discourse, which corresponds to the everyday situation that some amount of common knowledge is assumed, when a conversation begins.

¹² This time-stamping may not be so elegant; I believe it should be possible to get rid of it by using assumptions in the right way.



We can define a context in terms of a Prolog rule, which calls certain constraints and assumptions; here we show only the assumptions in the initial context.

```
duckville:-
  *acting(masc,huey),*acting(masc,dewey),
  *acting(masc,louie),    *acting(masc,donald),
  *acting(fem,daisy).
```

We can use it as follows, where it is applied to pronoun resolution.

```
| ?- duckville, phrase(discourse, [she,shoots,donald]).
event(0,shooting,daisy,donald),
dead(0,donald),
alive(0,daisy),
... ?
```

Note that this principle for setting up an initial context can also be used without the Hyprolog system, so that you can extend the examples shown in the previous sections (avoiding assumptions, of course, which is specific to Hyprolog).

5.3 More reading

The Hyprolog system is the result of my collaboration with Veronica Dahl [13,14], and I have made an implementation which is available at my website, <http://www.ruc.dk/~henning/hyprolog/> [9]; there are many examples here that may be useful to inspect. The assumptions in Hyprolog are inspired by Veronica's earlier work [19,34].

6 A few Prolog and CHR systems

There are several good Prolog systems around, some of which may be downloaded for free, but not all include CHR (and occasionally not even DCG).

All examples shown above run in both SICStus Prolog, www.sics.se/sicstus, and SWI Prolog www.swi-prolog.org. SWI is free, but SICStus costs money, although it is available in a test version for 30 days; check if your institution has a site license for SICStus.

You may find a list of all Prologs that support CHR at <http://www.cs.kuleuven.be/~dtai/projects/CHR/>.



References

1. Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.
2. Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing*. College Publications, 2006. Find also an online version at <http://www.learnprolognow.org/>.
3. Ivan Bratko. *Prolog (3rd ed.): programming for artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
4. Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, 1995.
5. CHR web. The programming language CHR, Constraint Handling Rules; official web pages. <http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/>
6. Henning Christiansen. Abductive language interpretation as bottom-up deduction. In Shuly Wintner, editor, *Natural Language Understanding and Logic Programming*, volume 92 of *Datalogiske Skrifter*, pages 33–47, Roskilde, Denmark, July 28 2002.
7. Henning Christiansen. Logical grammars based on constraint handling rules. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, page 481. Springer, 2002.
8. Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.
9. Henning Christiansen. HYPROLOG: a logic programming language with abduction and assumptions, 2005. Website with source code, User's Guide and examples, <http://www.ruc.dk/~henning/hyprolog/>.
10. Henning Christiansen. Logic programming as a framework for knowledge representation and artificial intelligence. Teaching note for the course “Artificial Intelligence and Intelligent Systems”, Roskilde University, Denmark, <http://www.ruc.dk/~henning/KIIS07/CourseMaterial/CourseNote.pdf>, 2006.
11. Henning Christiansen. Implementing probabilistic abductive logic programming with constraint handling rules. In Schrijvers and Frühwirth [32], pages 85–118.
12. Henning Christiansen. Executable specifications for hypothesis-based reasoning with prolog and constraint handling rules. *J. Applied Logic*, 7(3):341–362, 2009.
13. Henning Christiansen and Veronica Dahl. Assumptions and abduction in Prolog. In Elvira Albert, Michael Hanus, Petra Hofstedt, and Peter Van Roy, editors, *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL'04; At the 20th International Conference on Logic Programming, ICLP'04 Saint-Malo, France, 6-10 September, 2004*, pages 87–101, 2004.



14. Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2005.
15. Henning Christiansen and Verónica Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.
16. Henning Christiansen, Christian Theil Have, and Knut Tveitane. From use cases to UML class diagrams using logic grammars and constraints. In G. Angelova, K. Bontcheva, R. Mitkov, N. Nicolov, and N. Nikolov, editors, *RANLP 2007, International Conference: Recent Advances in Natural Language Processing: Proceedings*, pages 128–132. Shoumen, Bulgaria: INCOMA Ltd, 2007.
17. Henning Christiansen, Christian Theil Have, and Knut Tveitane. Reasoning about use cases using logic grammars and constraints. In Henning Christiansen and Jørgen Villadsen, editors, *Proceedings of the 4th International Workshop on Constraints and Language Processing, CSLP 2007*, volume 113 of *Computer Science Research Report*, pages 40–52. Roskilde University, 2007.
18. Alain Colmerauer. Metamorphosis grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–189. Springer, 1978. (Translation of an earlier report from 1975 in French: Les grammaires de métamorphose).
19. Verónica Dahl, Paul Tarau, and Renwei Li. Assumption grammars for processing natural language. In *ICLP*, pages 256–270, 1997.
20. Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer, 2002.
21. Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration*. Kluwer Academic Publishers, April 2000.
22. Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
23. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.
24. Thom W. Frühwirth. User-defined constraint handling. In David Scott Warren, editor, *ICLP*, pages 837–838. MIT Press, 1993.
25. Thom W. Frühwirth. Constraint handling rules: the story so far. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 13–14. ACM, 2006.
26. Jerry R. Hobbs, Mark E. Stickel, Douglas E. Appelt, and Paul A. Martin. Interpretation as abduction. *Artificial Intelligence*, 63(1-2):69–142, 1993.



27. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.
28. Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB*, pages 650–661. Morgan Kaufmann, 1990.
29. Robert A. Kowalski. *Logic for problem solving*. Elsevier North Holland, 1979.
30. Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer, 1997.
31. Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.
32. Tom Schrijvers and Thom W. Frühwirth, editors. *Constraint Handling Rules, Current Research Topics*, volume 5388 of *Lecture Notes in Computer Science*. Springer, 2008.
33. Swedish Institute of Computer Science. SICStus Prolog user’s manual, Version 3.12. Most recent version available at <http://www.sics.se/is1>, 2004.
34. Paul Tarau, Verónica Dahl, and Andrew Fall. Backtrackable state with linear assumptions, continuations and hidden accumulator grammars. In John W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, page 642. MIT Press, 1995.
35. Paul Tarau, Verónica Dahl, and Andrew Fall. Backtrackable state with linear affine implication and assumption grammars. In Joxan Jaffar and Roland H. C. Yap, editors, *ASIAN*, volume 1179 of *Lecture Notes in Computer Science*, pages 53–63. Springer, 1996.
36. David H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.



An Introduction to Natural Language Processing: the Main Problems

Veronica Dahl

School of CS, Simon Fraser University &
GRLMC-Universitat Rovira i Virgili
veronica@cs.sfu.ca

1 Definition, Scope

Natural Language Processing aims to give computers the power to automatically process human language sentences, mostly in written text form but also spoken, for various purposes.

This sub-discipline of AI (Artificial Intelligence) is also known as Natural Language Understanding. So first of all, what do we mean by “understanding”? According to Longman’s Dictionary of Contemporary English (1990), to understand is “to know or recognize the meaning of (something) or the words spoken by (someone).”

This definition plunges us into the ongoing debate about whether computers can know, think, feel, etc. We shall not attempt here to elucidate these deep questions. Instead, we will take the modest approach of acknowledging that no human or formal science has yet come up with a complete, satisfactory explanation of such human activities as thinking, knowing, and understanding.

However, we can still arrive at a useful working definition of what “understanding language” means for a computer, by noting that given a language stimulus, certain computer programs can replicate to some extent

some of the behavior that would typically be elicited by a human under the same language stimulus. Thus we can consider a computer program as being able to, in some sense, “understand” language, if its output given linguistic input roughly corresponds to a response that might be produced by a human given the same input.

For instance, consider a database query such as:

Do you know what time it is?

and consider the following possible answers:

1. *Five past eleven.*
2. *Yes.*
3. *True.*
4. *1.*

Assuming the time of the question is 11:05, the first is the kind of answer a human might produce, whereas the three remaining answers represent truthful but unhelpful versions of a literal answer: 2) replies in the affirmative, 3) gives a truth value representing the affirmative answer, 4) gives the same information in binary notation. We can think of these successive answers as diminishing in the degree of “understanding” shown, as measured by how close to human reaction the machine’s reaction is.

Similarly, if we ask a machine to translate:

‘La voiture n’a plus d’essence.’

we can conceivably get machine translations such as:

1. *The car has no more gas.*
2. *The car has no more essence.*
3. *The car has no more soul.*
4. *The car not has more of essence/soul.*

Here again, the degree of “understanding” evidenced by the translation given decreases as we proceed to the next example: “essence” is not the right word for translating ‘gas’, but is a better match than ‘soul’, whereas example 4) is simply a literal, word-by-word translation which does no justice to the original meaning.

This view of understanding language parallels the definition of intelligence proposed by Turing: if a machine can fool a human into believing he



is interacting with another human via a computer terminal, then it can be said to be intelligent.

But both Turing's test and the above proposed view of machine understanding of language have limitations. For instance, Weizenbaum's famous Eliza program, which modelled a psychological counselor, could certainly fool a human into believing s/he was interacting with another human via a computer terminal, yet it only did this through parrot-like, key-word oriented sentence transformations which could by no means be called "intelligent". For instance, a sentence of the form: *'I feel X'*, was simply transformed into *'Why do you feel X'*, with no regard whatsoever to what it might mean, through a purely word-processing transformation.

More modern language processing systems are still concerned with human-like behavior, but also take inspiration from the results that other disciplines relating to language and cognition have to offer. For instance, we can take a linguistic theory, and try to tailor it to our own computational needs, under the assumption that this theory comprises a usable if imperfect model of what might be actually happening in a human brain.

We are now ready to define the discipline and its scope.

Definition

Natural Language Processing is a branch of Artificial Intelligence in which computers are programmed to simulate an "understanding" of human languages such as Catalan, English etcetera, for various purposes, such as consulting databases through human languages rather than some specialized computer interface, or translating text automatically from one language to another, or communicating across virtual worlds.

Scope

The areas in which NLP has been most successful are text processing (vs. spoken language), for isolated sentences (vs. dialogue), and with restrictions regarding both the subset of language addressed and the problem domain. The results are, however, interesting within the domains addressed, and the language is restricted usually in ways which still result in allowing input one would naturally write, as opposed to condensed or telegraphic versions. For example the passive voice and a number of other linguistic constructs, yet those allowed do permit to express all concepts needed in a natural fashion.



Our notion of machine “understanding”, thus, while not being confined to behavior alone, is much more modest than that of human understanding: , it boils down to being able to respond to some extent as humans do, using adaptations of models of language provided or inspired by cognitive sciences, in particular linguistics. As pointed out in [5], “if machines are to communicate in human terms, they must embrace all the facets of natural language, and thus all of the parts of the broad topic of linguistics”.

We shall next discuss some of these facets, in order to understand some of the reasons why the mechanical understanding and translation of human languages is not as easy as one might expect upon first examination. In this process we shall see that, for good reason, most problems that are central to AI are also central to NLP, which makes NLP doubly difficult: it must embrace all parts of the broad topic of linguistics, as well as those of the broad topic of AI.

2 The most common problem in NLP

The most studied problem in natural language processing is the parsing problem: given a grammar and a presumed sentence in the language defined by that grammar, obtain some representative structure(s) if the sentence is indeed in the language. Whether for parsing or other NLP problems, such as generation, translation, concept extraction, etc, we need to capture an infinite number of sentences with a finite device such as a grammar. This implies the need for a concise, regularity-capturing description means, such as can be provided by logic programming.

Why is it so difficult? Concrete examples

Communications in natural language tend to assume vast contextual and empirical world knowledge that can be taken for granted in a human, but must be somehow spelled out for a machine. We shall examine a few cases in which this spelling out can be difficult.

Ambiguity:

In the first place, human language is plagued with ambiguities that we are not always aware of, owing to the fact that the knowledge of the world that



we have and unconsciously use often prevents us from even seeing alternative possible meanings that a computer program would have to consider.

Take for instance the database query:

Which is the price of a cabinet with four drawers?

While most humans will only see one meaning in this apparently straightforward question, a machine will have to decide whether '*with four drawers*' modifies '*price*' or '*cabinet*'. That is, are we asking for the price of a cabinet such that that price has four drawers, or for the price of a cabinet such that the cabinet has four drawers? Obviously (to us), the first meaning is nonsense, since prices cannot possess drawers. But for a computer, unless we have somehow programmed into it the world knowledge that protects us from even considering the nonsensical sense, both meanings are, a priori, equally likely.

A poignant example of structural ambiguity is the case of propositional phrases, where ambiguity is combinatorially explosive. Take for example the following sentence:

*I saw a man in the park with a telescope.*¹

This phrase can be interpreted in different ways according to which prepositional phrase attaches to which antecedent: either it tells us that the person was in the park and there saw a man who had a telescope, or that the person saw a man in the park and this park had a telescope, or that the person saw a man in the park through a telescope.

Different levels within Natural Language

We have no problem in simultaneously and effortlessly capturing the various levels speech involves (such as phonology, syntax, semantics, pragmatics, etc.), whereas precisely conveying them to a computer is quite difficult. In practice, most of the processing out there is syntactic, with some semantics to guide it. Speech recognition has lately advanced significantly, but still not enough to replace humans. The different levels can be characterized as:

Prosody: studies rhythm and intonation.

¹ The example belongs to Bill Wood



Phonology: studies sounds.

Morphology: studies syntactic subunits in a word (unity: a morpheme).

Syntax: studies rules for combining words into phrases and sentences.

Semantics: studies meaning.

Pragmatics: studies ways in which to use language with respect to world knowledge.

Note that none of these levels is trivial; e.g. according to how punctuation signs are placed in the following poem, we can find in it four completely different readings (Spanish speakers: find them!)

TRES BELLAS QUE BELLAS SON ²

Tres bellas que bellas son
me han exigido las tres
que diga de ellas cual es
la que ama mi corazón

si obedecer es razón
digo que amo a Soledad
no a Julia cuya bondad
persona humana no tiene
no aspira mi amor a Irene
que no es poca su beldad

Pragmatic Knowledge, Implicit Meanings:

Another difficulty with processing language, which we have hinted at with our example: *'Do you know what time it is?'* concerns the need for pragmatic knowledge of the world to be shared with a machine. This knowledge is largely implicit and even unconscious in humans, so it is not easy to think of all its possible instances ahead and spell them out.

² Cited by Roberto Vilches Acuña in: *Curiosidades literarias y malabarismos de la lengua*, Editorial Nascimento, Santiago de Chile, 1955.



A more radical example is the indirect request *'How cold it's getting'*, uttered in the hope that the hearer will in response close an open window he or she is near to. Whereas in the time request example there is at least a mention to the information wanted (what the time is), in this case, the utterance contains no hint whatsoever of what is actually being asked.

As a final example, consider:

I was caught running a red light and the pig fined me for it.

Most people would not have any difficulties understanding who *'the pig'* is, and that this is not actually about running but rather about driving a car past a red light. All this would be very hard for a computer to infer.

Imprecision

While natural languages are inherently imprecise, most methods for processing language are unable to properly deal with imprecision. Statistical parsing approaches view precision as a measure of how good a parsing result is and usually do not concern themselves with how to more precisely convey the meaning of an imprecise expression.

Zadeh [8] defines the notion of precisiation, within a theory called CW-Computing with Words-, as the conversion of a semantic entity, such as a question, proposition, command, etc. into another semantic entity which is computation-ready, that is, can be computed with.

For sentences that can be precisely interpreted, the notion of precisiation coincides with that of translating them into a semantic formalism which renders their interpretation and which can then be computed with by the usual means e.g. if the sentence is a query to a knowledge base, the evaluation of its semantic representation with respect to that knowledge base will compute into the answer to that query.

For sentences that cannot be precisely interpreted, existing machinery allows the precisiation of a simple imprecise sentence such as "Most Swedes are tall" such that it can serve as a basis for answering questions such as: "What is the average height of Swedes?" as concretely as possible, e.g. "Between approximately 170 cm and 200 cm" (also expressible by a user-friendly graphical interface provided by a specialized mouse called Z-mouse). However, this machinery is still being developed, and much needs to be done in terms of integrating it with contemporary NLU systems.



Coordinated sentences

Sentences with “and”, “or”, “but” and so on contain two or more sentences. They are a typical source of implicit meanings to be reconstructed, e.g. in: ‘*John ate a steak and drank a beer*’, we know right away that **John** drank a beer, even though the subject in the second conjoint is left implicit.

Reconstructing the missing parts is not as easy as in the above example unless we have adequate pragmatic information. For instance, almost unconsciously we realize that ‘*cold hands and feet*’ is shorthand for ‘*cold hands and cold feet*’, whereas no implicit repetition of ‘*cold*’ would even occur to us if we instead were to hear ‘*cold hands and fever*’.

Compound nouns

It is often difficult to see, in a sequence of several nouns, which of the nouns are head nouns and which are modifiers. For instance, Gerald Gazdar and Chris Mellish identify no less than 42 distinct structural descriptions in the innocent-looking phrase: ‘*Judiciary plea settlement account audit*’ [3], with just binary noun compounding. Many of these would not even occur to a human but must be carefully sorted through by a machine.

Overgeneration

Another problem is that of avoiding overgeneration. During analysis this is often not crucial, assuming the sentences that are input are correct, but for synthesis we should not allow our grammar to produce more sentences than the correct ones.

Long distance dependencies

Finally, we must provide a means for recognizing relationships between parts of the sentence that may be arbitrarily far away from each other, e.g. in order to relate a pronoun with the noun phrase it refers to, or in order to allow topicalization, as in:

Logic, we love.

Logic, I know we love.

Logic, I suspected he knew we love.

where the direct object of ‘*love*’ has been displaced for emphatic effect, and can appear at an arbitrary distance from it.



Stylistic resources

To complicate matters, stylistic resources such as metaphor, irony or allusion may also catapult a text beyond its literal meaning, thus making it difficult to know how much and which kind of background knowledge will be needed for a useful account of this meaning. Consider for instance the following excerpt from Juliet's Monologue, scene V, of Shakespeare's *Romeo and Juliet*:

Love's heralds should be thoughts, which ten times faster glide than the sun's beams, driving back shadows over lowering hills.

Clearly such texts are beyond literal interpretation: thoughts are being personalized- they do not in fact glide, except metaphorically; sun's beams do not '*drive*' shadows back, nor do the hills '*lower*'. Topicalization (the movement of a phrase outside from its more habitual order, for the purpose of giving it emphasis) is also present: '*ten times faster*' has been moved from where it normally would be located and now precedes the verb '*glide*'.

As an example involving humor, consider the following notice, found at a public service office (and from which I should take inspiration for charging my students ☺):

Answers: 1 euro

Answers that require thought: 2 euros

Correct answers: 4 euros

Discussions: 40 euros

Awkward smiles: free

Any human can immediately see the humoristic implications of this notice, but they would likely be lost to a computer.

Intention

Intention is paramount in human communication. Grice has uncovered principles of cooperative communication, and the usual assumption underlying NLP systems is that the intention is to communicate.

However, a message can be understood but stonewalled, as in the following dialogue between a new schoolteacher and one of the students in his class (example taken from the book *Le Petit Nicolas*):

Je m'appelle M. Leblanc- et vous? (My name is M. Leblanc- and yours?)
Nous non. (Ours isn't.)



It is obvious that the student knows the request is to know what their names are, rather than to know whether they are called M. Leblanc as well. The question's misinterpretation is intentional, and for a machine to analyze the reply's real meaning, it would need to be given very subtle contextual knowledge of beliefs and intentions.

The Cultural Dimension

This human reliance on world knowledge becomes even more difficult to emulate by a machine when different views of the world intervene. In the time request example, the knowledge of its implicit meaning is fairly universal. Almost anyone can interpret the meta-message contained in that literal message.

But when different cultures or world views intervene, interpreting meta-messages is not such a simple matter. Sociological research by [6], for instance, analyses how some meta-messages typical of one sex tend not to be understood by members of the opposite sex. These misunderstandings are explained in terms of the thesis that communication between the sexes is essentially cross-cultural communication, given that boys and girls grow up in what are essentially different cultures (manifested for instance through different kinds of games), and are socialized in different ways. Consequently, one sex leans more towards a hierarchical and problem-solving interpretation of the world, while another leans towards an interpretation based on relationships and networking. These differential upbringings result in different world views and a consequent difficulty in interpreting each other's meta-messages.

Now, if members of the same social group can have trouble understanding each other's meta-messages simply because of gender differences in upbringing, what can we expect for more patently cross-cultural communication? Many of us have direct experience of the problems encountered in a foreign country, even when our own language is spoken there. For instance, a reply of *'Thanks'* to a simple offer for a second helping of food may well mean *'Thanks, yes'* in one dialect, while meaning *'Thanks, no'* in another dialect of the same language. And even within the same country, or even city, we encounter language differences that can be quite marked, as in cockney versus the prescriptive norm, also known as "BBC English", or as in juvenile jargons vs. adult talk.

Rigorously speaking, then, all these pragmatic, stylistic, gender, dialectal, social class, and other differences would have to be encoded in some



computationally efficient way in order for a machine to be able to respond as subtly as a human. But of course, in machines we are content with much less than what we expect from humans. Typically, we reduce the domain of application in such a way that human-like reaction is made much easier. For instance, if we reduce ourselves to a hospital environment for a database consultation application, the subset of language to be considered, as well as the possible stylistic and other variants, reduces considerably.

In the next section we shall examine how much simplification we have been content with in the past, and how much more ambitious we can get given the state-of-the-art and the progress in hardware technology.

From all these examples we can see why it is so difficult to teach a computer to properly understand unrestricted language. It is a process that involves practically all aspects of human experience: thoughts, actions, feelings, beliefs, knowledge, expectations, time, learning, reasoning, metaphors, humor, irony, etcetera. Therefore, the central problems in Natural Language Processing include many of the central problems in Artificial Intelligence: how to represent knowledge, problem solving, reasoning, non-monotonicity, belief revision, planning, learning, ... in addition to its own specific problems.

3 Divide and Reign

Our problem being as formidable as we hope to have impressed upon the reader, the best hope we have of solving it is through dividing to reign, in particular by scaling it down to solvable size, and attacking different problems separately. This generates new sub areas within NLP.

For instance, one of the most widely studied areas in language understanding is that of question-answering systems, which usually model a subset of language specific to a given target domain (e.g. a medical domain).

While not easily adaptable to other domains, these systems have met with reasonable success, owing to the fact that the context of discourse in these systems can be largely predicted. This is useful for instance in resolving ambiguities due to polysemy (multiple meanings for one word). In a query system for a financial domain, for example, we could describe only one meaning- the most likely one- of the word '*bank*' (as a financial institution), on the reasonable assumption that most users of this system will use the word in its financial, not its geographic (as in '*the bank of a river*') sense.



We can also reduce the range of natural language to be accepted (for instance, only sentences in the active, not passive voice, will be allowed).

A further simplifying factor concerns the limitation to single, isolated queries, which allows us a smaller context in which to look, for instance, for pronoun reference.

Machine translation is a sub-area that has had a more eventful history. Its initial period was characterized by the naive view that one-to-one correspondences existed between languages, and all we had to do was to encode them. But for instance, languages such as Lithuanian or Russian do not have any articles. The colors that are recognized in a rainbow vary from one culture to another. The one-to-one correspondence view soon collapsed under the evidence that many of the concepts and the syntactic and lexical constructions used to cover them are language or culture dependent. The initial over-enthusiasm resulting from the naive view gave way to an equally excessive pessimism, and funds became very scarce.

The three main approaches to machine translation are the direct one, the rule-based one, and the inter-lingua, with statistical approaches pitching in as well [4].

In the direct approach, heavily language-specific programs are designed to translate from one given language into another. Word-by-word replacement routines are complemented with ad hoc transformations performed after lexical substitution.

In the rule-based, also called transfer, approach, a source-language sentence is translated into a machine-readable form corresponding to the source language. This form is then mapped into a machine-readable form for the target language, and then the output is generated from it. The intermediate, machine-readable forms are dependent on the language considered. Therefore, mappings need to be constructed for each source-language/target language pair.

In the inter-lingua approach, the source-language sentence is mapped into a language-independent representation, from which the surface structure is systematically produced.

Just as natural language query systems do, successful machine translation systems also restrict their scope to very limited domains of discourse (e.g. weather reports), and to specific subsets of natural language (e.g. the relatively unambiguous declarative sentences found in technical documents). As well, they usually depend on interaction with human users



and/or post-processing by a human translator in order to correct the system's errors.

Machine translation systems also usually pay the price of extensive development and tuning to particular clients. To date, these systems are not yet widely used, both because of the limitations described above, and because of the time and expense required to develop them. So for a translation system we might relax our NLP definition even further and suggest that, even if its output is not what would be produced by a human, it will be acceptable if it helps the job of a human translator, e.g. by reducing his/her job to correcting the system's mistakes.

4 Concluding remarks

We have discussed some of the main difficulties in formalizing and automating the tasks of processing human language. As we have seen, some of them are even hard to imagine for humans who are used to effortlessly using and interpreting language, since so much of the knowledge used in so doing is unconscious and resorts to fuzzily defined world knowledge.

One implicit notion throughout our discussion is the desirability of using the results of cognitive sciences as well, and in particular of linguistic theory. Theoretical linguistics has indeed developed remarkable insights on some very complex linguistic phenomena, and is developing in directions which are more and more compatible with computational linguistic needs. However, it is also fair to observe that these theories do not have as their goal or method to provide immediate comprehensive descriptions of actual natural language. A natural language processing system that must process actual text (e.g. spontaneous speech) with a minimum of coverage and accuracy, must solve many problems for which linguistic theory does not have even in principle solutions. Moreover, when building actual language processing systems, many instances are found in which the analyses of linguistic theory are contradicted by the data.

The main challenge is, then, to adapt the general analyses and insights from linguistic theory into actual language processing systems, and to delicately interweave the many independently explored facets of language processing.




Acknowledgments

Support from the European Commission in the form of the author's Marie Curie Chair of Excellence, and from both the Universitat Rovira i Virgili and the Simon Fraser University, as well as from the Canadian National Sciences Research Council, is gratefully acknowledged.

References

1. Allen, J. (1994) *Natural Language Understanding*. Benjamin Cummings, Second Edition.
2. Covington, M. (1994) *Natural Language Processing for Prolog Programmers*. Prentice-Hall.
3. Gazdar, G. and Mellish, C. (1989) *Natural Language Processing in Prolog*. Addison-Wesley.
4. Koehn, Ph. (2010) *Statistical Machine Translation*. Cambridge University Press, 2010.
5. McEnery, C. L. (1992) *Computational Linguistics: A handbook and toolbox for natural language processing*. Sigma-Press. Wilmslow, U.K., 1992.
6. Tannen, D. (1990) *You just don't understand- women and men in conversation*. Balantine Books, New York.
7. Jurafsky D. and Martin, J. (2009) *Speech and Language Processing- An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Second Edition. Prentice-Hall.
8. Zadeh, L. (2010) *Computing with Words- A Paradigm Shift*. UC Davis CS Colloquium, January 7 2010.





TRIANGLE is a periodical published by the Department of Romance Studies. It aims to present the results of interdisciplinary research which adopts new approaches to understanding language sciences.

The first issue, **Introduction to language and computation**, is a collection of presentations given at a workshop for humanities students at the URV during the 2009–2010 academic year. The issue aims to give a comprehensive view of how computation can help process and understand natural language.

