LOGIC PROGRAMMING AND LANGUAGES

The connection between logic and language has been apparent for long, and much has been written about the marvel and wonder of human language, that intricate tool for communicating thoughts, feelings, and emotions, for expressing human nature and society, and for transmitting and clarifying knowledge and belief. Unlike all other tools used by humankind, it is also, in a very real sense, a living creature in constant change.

Only since the development of logic as a programming language circa 1972 (1), however, has the relationship between

logic and language developed to the point at which we can describe a grammar roughly in terms of logic, and let a hidden theorem prover answer for us a variety of interesting questions about that language. Such questions include, for instance Is sentence X in the language defined by the grammar? What is the meaning representation of sentence X with respect to a given grammar? What sentence can be obtained from meaning representation Y with respect to a given grammar? What is the translation into German of the English sentence X?

The attraction of using logic programming for processing language is threefold. In the first place, many of its features are naturally adapted to deal with natural language processing needs. Knowledge about language, in many linguistic frameworks, is expressed as some sort of deduction from general principles which can be expressed as axioms. The aims of conciseness and generality are also common to both modern linguistics and logic programming, and the declarativeness inherent in logic programming is also one of the aims of modern linguistics. In the second place, logic provides a natural underpinning for natural language semantics (meaning). Thirdly, different types of logics have played important roles in linguistics and in natural language processing. By choosing a logic-based formalism to process language, the implementing means become closer to some of the presentations manipulated in processing, thus minimizing the need for interfaces by providing a uniform methodology, logic throughout, in one form or another. Formal characterizations of the logic programming tools developed can also be done in terms of some kind of logic.

In what follows, through logic programming, we provide an intuitive explanation of many of the concepts and techniques involved in natural language processing.

LOGIC PROGRAMMING

Logic programming is the art of describing a problem domain in terms of logic clauses plus some extralogical features (e.g., input/output, control) that render such a description executable by a logic programming processor, e.g. Prolog. Execution is triggered by a query (representing a specific problem in the problem domain described by the clauses). Automatic deduction takes place from the query and the clauses defined, through resolution-based theorem proving (2). In all that follows, we shall use Prolog notation because it is the most common programming language.

Clauses have the general form

$$p : p_1, \ldots, p_n$$

where p and the p_i are predicates, all variables in the clause are assumed to be implicitly universally quantified, "," stands for "and, and ":-" stands for "if". Thus if the clause contains variables $X_1, \ldots, X_n$, it is read: "for all values of $X_1, \ldots X_n$, p is true if p_1 and . . . and p_n are all true.

Conditionless clauses are called assertions and noted as p.

Here is for instance a complete Prolog program for the problem domain of family and liking relationships among dogs:

```
likes(rover,mimi).
likes(rover,light).
likes(Y,X) :-mother_of(X,Y).

mother_of(rover,light).
mother_of(mimi,night).
```

Some conventions: variables are noted by identifiers starting with a capital letter; constants are denoted by lower case identifiers; underscore is used for composite predicate names. Thus the above clauses express, from top to bottom, "Rover likes Mimi", "Rover likes Light", "For every X and every Y, Y likes X if Y is the mother of X", "the mother of Rover is Light", and "the mother of Mimi is Night".

Queries have the form

$$? = p_1, \ldots, p_n.$$

where again, the p_i are predicates, "," stands for "and", and "?-" is interpreted as a request to find values for the variables in the query, if such exist, which make the conjunction p_1 and . . . and p_n true.

For instance, with respect to the above program, we can query:

```
?-likes(X,Y).                    (Who likes who?)
```

to which Prolog will respond:

```
X = rover, Y = mimi;
X = rover, Y = light;
X = light, Y = rover;
X = night, Y = mimi;
no                               (no more answers)
```

Another sample query and answer follow:

```
?-likes(X,rover),likes(rover,X).   (Who likes and is liked by Rover?)

X = light;
no                               (no more answers)
```

A predicate's arguments are logic terms, that is, either constants (e.g., light), variables (e.g., X), or functional expressions of the form $f(t_1, \ldots, t_k)$, where the t_i are in turn terms. For instance, the third clause in our program above could have been expressed using a functional expression "mother(X)" to represent the individual who is the mother of X:

```
(likes(mother(X),X).
```

Whatever convention we choose (functional or relational notation to represent motherhood), we have to stick to it throughout our program. If we use functions as arguments, it is important to take into account that functions in Prolog do not evaluate: they take values through unification (the process by which variables get assigned values during resolution).

Because our focus in the rest of this article is on a syntactic variant of logic programming, namely, logic grammars, no

more details of Prolog are given here. The interested reader can consult one of the many Prolog textbooks in existence.

LOGIC GRAMMARS

A grammar is a finite way of specifying a language which may consist of an infinite number of sentences. A logic grammar (3) consists of what are called "rewrite rules," which use logic terms as symbols. A rewrite rule has the form

$a \rightarrow b$

and expresses that a can be replaced by b. When we use logic terms as symbols, as in logic grammars, it expresses the more powerful statement that something *of the form a* can be replaced by something *of the form b*. "Of the form" is interpreted as *literally identical* or as *amenable to a literally identical form* by substituting terms for variables. For instance, the following rewrite rule, which contains no variables,

name → [popocatepetl].

expresses that if you have "name" you can replace it by "popocatepetl" with no further ado. Notice that words in the language we are defining (e.g., "popocatepetl") are noted between square brackets. Technically, they are called *terminals*. Words that denote grammatical components, such as "name", "verb", and "adjective", are technically called *nonterminals*. In Definite Clause Grammars, the Prolog most common embodiment of logic grammars, terminals never appear in a left-hand side. Variables are denoted by identifiers starting with a capital, whereas constants (proper names) start with lower case, by convention.

The rule

verb(X,Y, loves(X,Y)) → [loves].

on the other hand, expresses that if you have something *amenable to the form* verb(X, Y, loves(X, Y)), you can replace it by "loves". For instance, if you have

verb(eve,popocatepetl,P)

this is amenable to the form in the left-hand-side of the rule simply by substituting X=eve, Y=popocatepetl, and P=loves (eve,popocatepetl). Notice that substitutions affect all occurrences of a variable in the rule. For example, where P is replaced by loves(X,Y), it becomes loves(eve,popocatepetl), because X and Y are replaced, respectively, by "eve" and "popocatepetl".

In this way, by using rewrite rules, we construct a complete grammar for a language, so that sentences in the language are recognized as such, or even analyzed into some desired representation, as a side effect of querying the complete grammar given a Prolog processor. We can, for instance, complete the rules above into the full grammar:

name(eve) → [eve].
name(popocatepetl) → [popocatepetl].

verb(X,Y,loves(X,Y)) → [loves].

sentence(P) → name(X),verb(X,Y,P),name(Y),

and then query this grammar through a call to a predefined Prolog binary predicate "transform" (easy for a Prolog programmer to define, but the details of which shall not concern us here), whose first argument is the initial symbol of the grammar and whose second argument is the sentence to be analyzed, written as a Prolog list (i.e., as a list of words separated by commas and enclosed in square brackets). Queries in Prolog are preceded by "?-". The query

?-transform(sentence(P),[eve,loves,popocatepetl]).

for instance, will produce the answer:

P = loves(eve,popocatepetl).

The rewritings and substitutions involved in Prolog (invisibly) obtaining this answer are as follows:

$$\begin{aligned}\text{sentence}(P) &\rightarrow \text{name}(X), \text{verb}(X, Y, P), \text{name}(Y)\\ &\rightarrow [\text{eve}], \text{verb}(\text{eve}, Y, P), \text{name}(Y) X = \text{eve}\\ &\rightarrow [\text{eve},\text{loves}], \text{name}(Y) \rightarrow [\text{eve},\text{loves},\text{popocatepetl}]\\ &\quad P = \text{loves}(\text{eve},Y) \qquad Y = \text{popocatepetl}\end{aligned}$$

The same grammar can be used, of course, to analyze "popocatepetl loves eve". The answer in this case is P=loves(popocatepetl,eve).

Notice that the final value for P is obtained by composition of the various substitutions used (i.e., P=loves(eve,Y) and Y=popocatepetl). Another interesting thing to notice is that the meaning representation "loves(eve,popocatepetl)" is incrementally obtained. Initially, it is just a variable. The verb rule makes it further known as a "skeleton" of the form loves (X,Y), but because X has taken a value, upon applying the verb rule, it becomes further known as loves(eve,Y) until the last call, name(Y), uncovers Y's identity.

SYNTAX VS SEMANTICS

It is useful to separate the notions of what form a sentence has (e.g., a proper name followed by a verb followed by another proper name) from what the sentence means. Matters relating to form are the domain of syntax. Those relating to meaning are the domain of semantics. The value obtained for P in the grammar of the previous section is called a *meaning representation* or a *semantic representation* for the sentence. Many different semantic representations are possible. For instance, if we wanted to obtain a hyperbolic generalization of the sentence "eve loves popocatepetl", for example, "adores(generic_woman, volcanoes)" as a meaning representation, all we have to do is modify the rules of the grammar as follows:

name(generic_woman) → [eve].
name(volcanoes) → [popocatepetl].

verb(X, Y, adores(X, Y)) → [loves].

sentence(P) → name(X), verb(X, Y, P), name(Y).

More generally, meaning representation formalisms other than first-order logic form can be used (e.g., conceptual structures, semantic networks, scripts, situation semantics, Montague semantics, etc.).

GRAMMAR REVERSIBILITY

In more complex grammars than this one, reversibility (i.e., using the same grammar for generating as well as analyzing sentences) is usually not possible because of practical concerns outside the scope of this article. It is interesting to note, however, that for this grammar we can generate a sentence from its internal representation, and thus obtain reversibility, simply by querying, for instance,

?-transform(sentence(loves(popocatepetl,eve),S).

The result obtained is

S = [popocatepetl,loves,eve].

LANGUAGE TRANSLATION

Translation grammars are also easily prototyped. If we merely add one more argument to the grammar symbols to keep track of the language we are in, we can produce the following bilingual grammar:

name(eve,L) → [eve].

name(popocatepetl,L) → [popocatepetl].

verb(X, Y, loves(X, Y), english) → [loves].
verb(X, Y, loves(X, Y), french) → [aime].

sentence(P, L) → name(X, L), verb(X, Y, P, L), name(Y, L).

Because we do not have accent characters in the terminal, the name rule for "eve" is the same in both languages (as it is for "popocatepetl", given that it is the same in both French and English). This is indicated by leaving the language argument as a variable (L), which attracts to it the proper value—"english" or "french"—according to the context. Context is provided by the sentence rule, which requires that the same value L be used for the subject name, the verb and the object verb to have a sentence in the language L.

Notice that, for the meaning representation of an (English or French) word, we use an English mnemonic name (e.g., "loves" rather than "aime," in the rule for "verb"), because we need the same meaning representation for both languages, so that we can go from one to the other through this internal representation.

Now, we can query, for instance,

?-transform(sentence(P),[eve,loves,popocatepetl],english),
transform(P, S, french).

The first call to "transform" associates P with the value "loves(eve,popocatepetl)," and this value of P is input to the second call to "transform," which then generates from it the sentence S= [eve,aime,popocatepetl].

LINGUISTIC THEORY

The examples we have seen so far are, of course, quite simplified to be clear to the uninitiated. "Real life" language processing has to deal with all the formidable complexities found in natural languages. Linguistic theories offer useful but incomplete organizing frameworks for dealing with this complexity and by no means agree on all points. In addition, the activity of *computationally* processing language often stresses aspects different from those stressed by linguists. Thus, language processing problems, by necessity, are addressed without clear guidance from formal linguistics. In adapting various pure linguistic theories for computational use, interesting mutual feedback between formal and computational linguistics ensues.

Thus it is important to take into account the most general analyses from linguistic theory in building natural language processing systems, while also trying to adapt and combine the different theories to our ends, given that no single one can offer all-encompassing solutions.

The transformational of generative paradigm (4) provided an initial step towards *computationally* usable linguistic models by viewing grammars as highly formalized entities. These entities consisted of two components, a base component of context-free rewriting rules (i.e., like our rewriting rules minus the symbol arguments, i.e., minus unification as well), which described a "canonical" version of sentences (i.e., in the active voice, affimative form, etc.) and a transformational component, which contained general rules to convert these canonical representations into other possible variants (passive voice, interrogative or negative form, etc.).

Although the most formalized linguistic paradigm until then, transfomational theory was not easily amenable to computational treatments, mainly because of the myriads of specific rules engendered. New theories emerged, all with the objective of brevity of description in mind. Lexical Functional Grammar (5) born under the explicit goals of computational preciseness and psychological realism, replaces transformations by dealing with them in the lexicon. Generalized Phrase Structure Grammars (6) aimed at succinctness by providing higher level grammars that could be mechanically converted into context-free grammars. Categorial grammars analyze language expressions as the functional product of a functor applied to a suitable set of simpler argument expressions (7). The categorial grammar approach lends itself very nicely for studying the relationship between the syntactic structures and the semantics of language expressions. All of these linguistic models strive in different ways for the same objectives of principledness and succinctness and, in so doing, have developed similarities between themselves and also with logic programming. As an example, some notion of unification is also present in most contemporary linguistic models, although less crucially than in logic programming.

Despite considerable progress by modern linguistic theories toward formalized accounts of human language, their adaptation for computational use remains difficult, for reasons such as the following:

- Modern linguistics stresses competence (the tacit knowledge that a speaker has of the structure of his/her language) over performance (how language is processed in real time, why speakers say what the say, how language is used in various social groups, etc.), whereas the latter considerations are more prominent in building natural language systems.
- Linguistic efforts to account for competence, particularly in the past, yielded mostly explanations of language

synthesis, whereas computational linguistics is often more interested in analyzing language than in synthesizing it. Although this is changing, because modern linguistic theories are intent upon declarativeness and lack of bias towards one processing direction, the change is not as swift as would be desirable.

Formalizations of linguistics to the point that it is conceivable to use them for automatic processing are relatively recent and constantly evolving.

Thus, natural language processing is still an art, whose intersection with logic programming is that of two highly promising and complementary, but also rapidly changing scenarios. Cross-fertilization with each other and with other fields is only to be expected and is indeed happening.

APPLICATIONS

Some of the systems developed around specific applications aim mainly to advance the state of knowledge, whereas others aim at carrying out practical natural language processing tasks. For the former, elegance and theoretical basis are paramount, whereas the latter are mostly concerned with coverage and efficiency, although both types of systems, of course, exhibit both kinds of concerns to a certain extent.

The theoretically oriented systems include those which make extensive use of linguistic theory, such as Fong's (8), which correctly accounts for hundreds of different constructions from an introductory linguistics textbook, and those which develop new representational devices.

Many other specific applications have been explored. For instance, logic grammar aided the following: learning of lexicons; detecting grammatical mistakes of a student learning a second language; assisted applying sentence compositions to language interfaces; applications for communicating with handicapped persons; machine translation for agricultural reports; and reversible language processors (those which are easily adapted both for analysis and for synthesis).

Let us also mention that some of the formalisms developed with computational linguistics in mind have found applications outside it. For instance, DCSGs (9), a logic grammar formalism for free word order languages in which grammar rules are viewed as definitions for set conversions, also has applications to general problem solving. And DCTGs, an extension of logic grammars, in which the construction of semantic representation is modularized and semiautomated (10), are used in software specification problems.

We find ourselves at the exciting historical point where the advances of logic programming make it possible to address the needs of growingly ambitious applications in natural language processing with hopes of reasonable efficiency and where theoretical linguistics are coincidentally developing in directions more and more compatible with the needs of computational linguistics.

BIBLIOGRAPHY

1. A. Colmerauer et al., *Un Systeme de Communication Homme-Machine en Francais,* TR, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, Marseille, 1973.
2. J. A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM,* **12**: 23–24, 1965.
3. A. Colmerauer, Metamorphosis Grammars, in *Lecture Notes in Computer Science,* New York: Springer-Verlag, 1978.
4. N. Chomsky, *Lectures on Government and Binding,* Dordrecht, Holland: Foris Publications, 1981.
5. J. Bresnan, *The Mental Representation of Grammatical Relations,* Cambridge, MA: MIT Press, 1987.
6. G. Gazdar et al., *Generalized Phrase Structure Grammar,* Cambridge, MA: Harvard University Press, 1985.
7. R. T. Oehrle, E. Bach, and D. Wheeler (eds.), *Categorial Grammars and Natural Language Structure,* Dordrecht, The Netherlands: Reidel, 1988.
8. S. Fong, Computational Properties of Principle-Based Grammatical Theories. Ph.D. Thesis, MIT Al Lab, 1991.
9. T. Tanaka, Definite-Clause Set Grammars: A Formalism for Problem Solving, *Journal of Logic Programming* **10** (1): 1–18, 1991.
10. H. Abramson, Definite Clause Translation Grammars, *Proc. IEEE Logic Programming Symp.,* Atlantic City, NJ, 1984.

VERONICA DAHL
Simon Fraser University

LOGIC, SUPERCONDUCTING. See SUPERCONDUCTING ELECTRONICS.