Declarative Programming: Handout 4

Yılmaz Kılıçaslan
October 25, 2023

Key Concepts: *Data Structure, Control Structure, (Logical) Term, Compound / Non-Compound Term, Constant, Variable, Atom, Integer, Floating Number, Functor, Arity, Property, Relation, Procedure, Iterative Clause, Quantification, Domain of Discourse, Syntax, Semantics, Instantiation, Generalization, Ground / Nonground Terms, Mental State, (External) Situation, Epistemological Level.*

LOGIC PROGRAMMING / PROLOG IN FIRST-ORDER LOGIC

1 INTRODUCTION

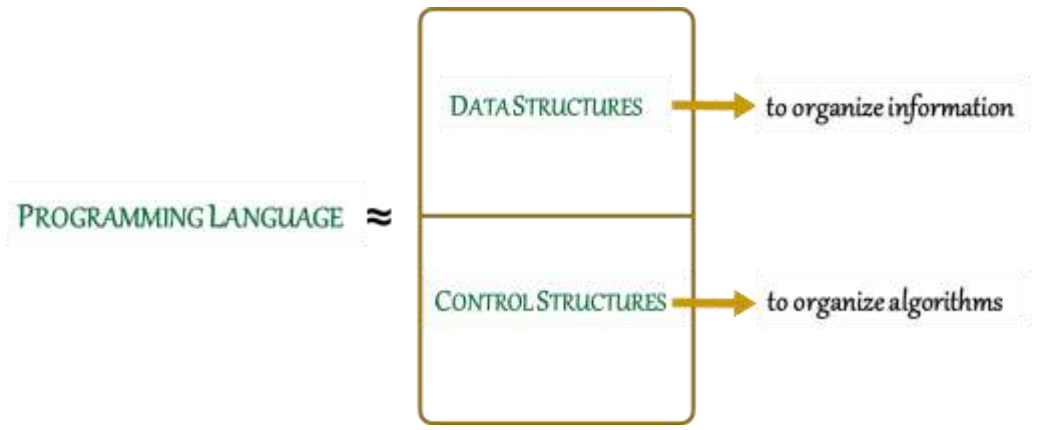
Last week, we made an introduction to logic programming and Prolog. We restricted our inventory of syntactic structures to that of propositional logic. This was practically weird. Probably no one would try to write a down-to-earth logic program with a power of expression confined to propositional logic. Our aim when doing this was purely pedagogical. We wanted to concentrate on the types of statements used in logic programming.

Nonetheless, remember that we pointed out a few times that Prolog gains its full power when it uses the structures of first-order predicate logic. In fact, Prolog was designed to cover a subset of that logic system, which goes well beyond the power of propositional logic. Today, we will see how that subset is implemented in logic programming / Prolog.

2 TERMS: THE SINGLE DATA STRUCTURE IN LOGIC PROGRAMMING

As is well-known, a programming language comes with **data** and **control structure**s to use when writing programs:

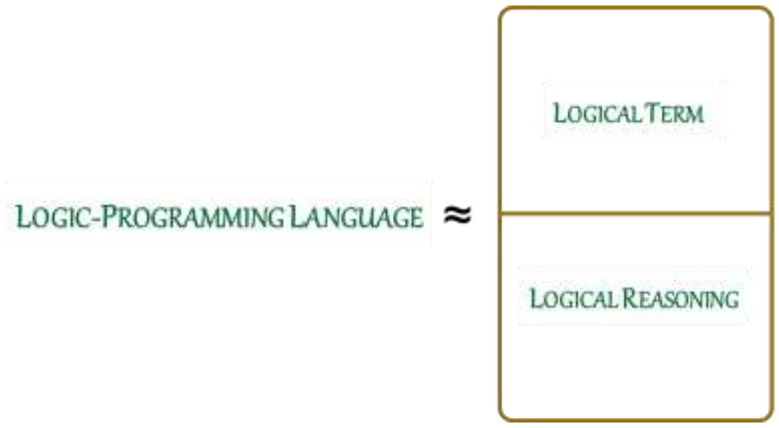
(1) PROGRAMMING LANGUAGES IN GENERAL:



In general terms, data structures organize information, whereas control structures organize algorithms.

At a purely declarative level, control in logic programming amounts to logical reasoning:

(2) LOGIC-PROGRAMMING LANGUAGES:

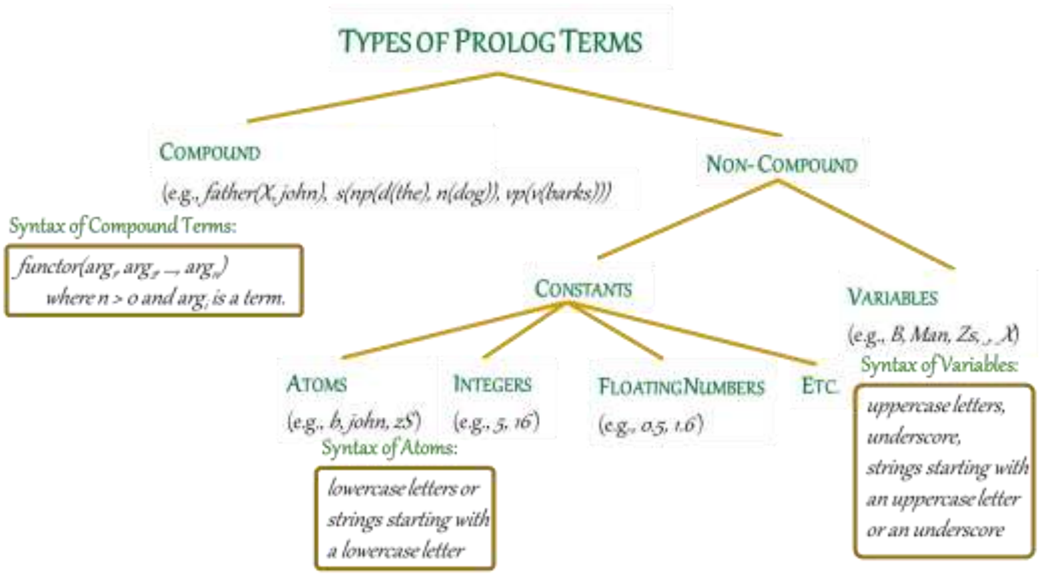


Albeit not completely so, for the time being we assume that programming with Prolog is at this pure logical level. We will return to the ways in which Prolog deviates from pure logical control in the coming weeks.

There is a single data structure used in logic programming: the **logical term**. Today, we will focus on the syntax of logical terms used in Prolog.

Terms in Prolog can be either **compound** or **non-compound**. Non-compound terms are further divided into two categories: **constant**s and **variable**s. Constants include **atom**s, **integer**s, **floating number**s, etc.:

(3)

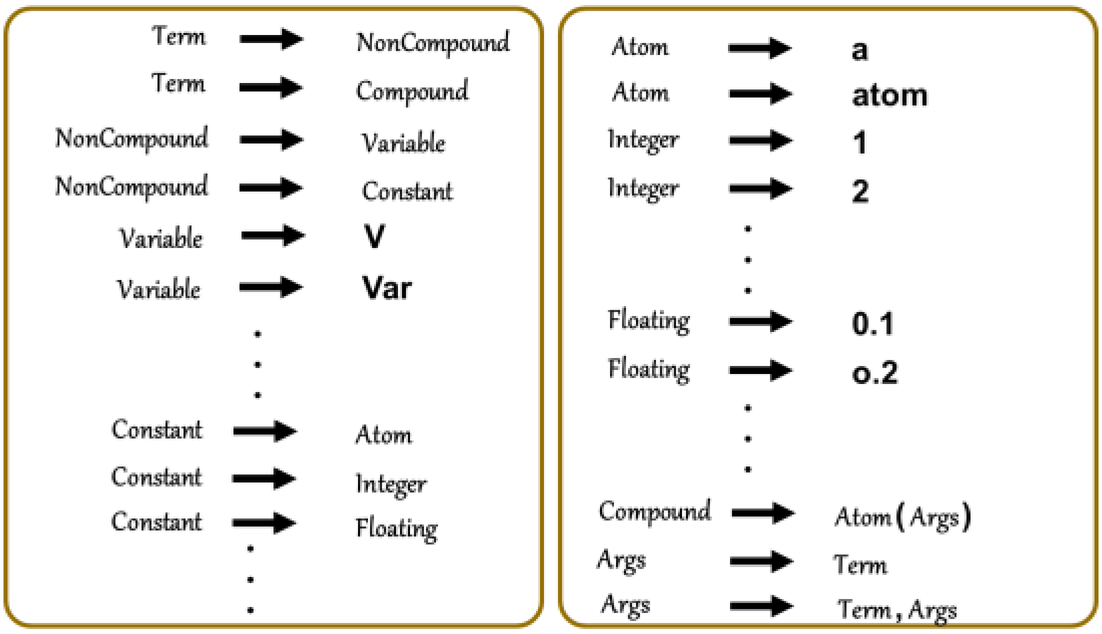

TYPES OF PROLOG TERMS
COMPOUND
(e.g., father(X, john), s(np(d(the), n(dog)), vp(v(barks))))
Syntax of Compound Terms:
functor(arg1, arg2, ..., argn) where n > 0 and argi is a term.
NON-COMPOUND
CONSTANTS
VARIABLES
(e.g., B, Man, Zs, _, X)
Syntax of Variables:
uppercase letters, underscore, strings starting with an uppercase letter or an underscore
ATOMS
(e.g., b, john, zS)
Syntax of Atoms:
lowercase letters or strings starting with a lowercase letter
INTEGERS
(e.g., 5, 16)
FLOATING NUMBERS
(e.g., 0.5, 1.6)
ETC.

As a syntactic convention, variables are either uppercase letters or strings that start with an uppercase letter and, in this way, are distinguished from constants (e.g., *B* vs. *b* and *Man* vs. *john*). It is worth noting that Prolog has also *anonymous* variables in its inventory. They are variables that are supposed to appear only once in a clause and, hence, need not be named from an implementation point of view. Prolog adopts the convention of using an underscore or a string starting with an underscore as the name of an anonymous variable (e.g., _, _X).

Compound terms are recursively defined as predicate-argument structures. Predicates are technically referred to as **functor**s and are taken to denote either properties or relations. Like atoms, they have to be either lowercase letters or strings starting with a lowercase letter. The number of the arguments of a functor gives its **arity**. When referring to a functor, the common practice is to write it with its arity in the format *functor/arity* (e.g., *father/2*, *n/1*). As you know, **properties** are unary predicates and **relations** can be binary, ternary, quarternary, etc. In fact, we can even have 0-ary predicates, which correspond to atomic propositions in propositional logic.

As for arguments of a functor, they can be any other terms, including compound ones. Hence, we have a recursive structure (e.g., *s(np(d(the),n(dog)), vp(v(barks))*). Below is a (partially specified) BNF grammar for terms:

(4) A FRAGMENT OF A BNF GRAMMAR FOR PROLOG TERMS:



Term	→	NonCompound	Atom	→	a
Term	→	Compound	Atom	→	atom
NonCompound	→	Variable	Integer	→	1
NonCompound	→	Constant	Integer	→	2
Variable	→	V		.	
Variable	→	Var		.	
	.			.	
	.		Floating	→	0.1
	.		Floating	→	o.2
Constant	→	Atom		.	
Constant	→	Integer		.	
Constant	→	Floating		.	
	.		Compound	→	Atom(Args)
	.		Args	→	Term
	.		Args	→	Term, Args

Let us now see how the three types of statements (i.e., facts, rules, and queries) are structured using compound terms in Prolog.

3 THE USE OF COMPOUND TERMS IN PROLOG STATEMENTS

3.1 Facts

Facts can be used to state that a property is borne by an object:

(5) *female(sarah).*

or that a relation holds between objects:

(6) *mother(sarah, john).*

Remember that syntactically each statement is required to be followed by a dot. Needless to say, functors (i.e., predicates) must start with lowercase letters, as they are not allowed to be variables. Can you explain the reason behind this restriction?

3.2 Rules

Rules can be used to make a generalization. A sentence like "if Mary is happy, then John is happy" can translate into Prolog as follows:

(7) *happy(john):- happy(mary).*

A collection of rules with the same functor in the head is called a **procedure**. Below is a *happy/1* procedure:

(8) *happy(john):- happy(mary).*
happy(marry):- love(john, marry), not(hungry(mary)).

A rule with one goal in the body, such as the first one above, is called an **iterative clause**.

3.3 Queries

Recall that queries are a means of retrieving information from the program and appear in a context different from that containing the program:

(9) *?- mother(sarah, john).*

The query above asks whether the *mother* relation holds between *sarah* and *john*.

It is worth re-emphasizing that answering a query with respect to a program is determining whether the query is a logical consequence of the program. Therefore, interpreted logically the query above amounts to a question like "Does the proposition mother(sarah, john) necessarily follow from the given program?" or, equivalently, "Can we prove it from the program?", even though we can still interpret it more naturally as "Is Sarah John's mother?".

4 QUANTIFICATION IN PROLOG

4.1 Quantification in Program Statements

Note that the examples we have seen in the preceding section are quantifier-free propositions (i.e., they do not contain variables). Therefore, they can be thought of as examples of propositional logic (at most, of one in which propositions are analyzed into their predicates and arguments). Recall from the Formal Logic course that we need individual variables (i.e., variables that take individuals as values) and quantifiers over these variables in order to get to first-order (predicate) logic (FOL), which is a logical system stronger than propositional logic.

Prolog has both individual variables and quantifiers. (Non-anonymous) variables, as we saw in Section 1, are encoded explicitly with uppercase letters or strings starting with an uppercase letter. Quantifiers, on the other hand, do not appear on the surface of the code. All variables in prolog statements are implicitly quantified, existentially or universally. The type of quantification is determined by the context in which the quantified variable appears.

Variables appearing in program statements are universally quantified. Suppose that all the individuals in our domain know French, then one way to state this will be to add the following fact to the program:

(10) *know(X, french).*

When used in this way, variables may serve to summarize many facts. For example, suppose that our domain of discourse includes *Ali, Mehmet, Oya,* and *Suna,* then the fact above can be thought of as a summary of the following list of facts:

(11) *know(ali, french).*
know(mehmet, french).
know(oya, french).
know(suna, french).

In more precise words, the fact in (10) is synonymous with the following first-order logic sentence:

(12) ∀*x know(x, french)*

However, notice that full synonymy between (10) and (12) is not something we would like to see in this case. In its intended meaning only the individuals in the domain of discourse (*Ali, Mehmet, Oya,* and *Suna*) would be stated to know French. But, (12) states that everyone and, more unwantedly, everthing knows French. In fact, the syntactic structure of this formula reveals the problem in its semantics. Though legitimate, the syntax of (12) does not conform to those we saw in our logic course. Recall that all universally quantified sentences are expected to come with an implication operator (i.e., →) used as the main operator in the scope of the quantifier. Recall also that the antecedent of this operator in universally quantified formulae serve to restrict the domain of quantification. Whatever is stated is stated about this restricted domain. Therefore, a better formulation of the meaning intended with the example above can be the following:

(13) ∀*x [person(x) → know(x, french)]*

with the intended denotation of *person* being confined to the domain of discourse. When it comes to prolog, the same meaning can be encoded with the following rule:

(14) *know(X, french) :- person(X).*

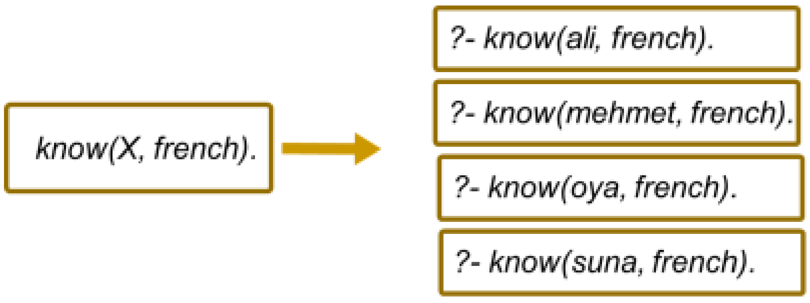
along with the following facts:

(15) *person(ali).*
person(mehmet).
person(oya).
person(suna).

In the previous lecture, we exemplified only two of the deduction rules of Prolog, namely, the rule of *identity* and that of *Modus Ponens*. The other two deductions rules, **instantiation** and **generalization** are defined on quantified statements.

Any instance of a universally quantified fact can be deduced from it. This is what we call the deduction rule of instantiation. For example, from the fact in (10) we can deduce any of the facts in (11):

(16)



4.2 Quantification in Queries

While variables in programs are supposed to be universally quantified, those in queries are taken to be quantified existentially. For example, the query

(17) *?- know(X, french).*

reads: “Does there exist an X such that X knows french?”, which translates into FOL as:

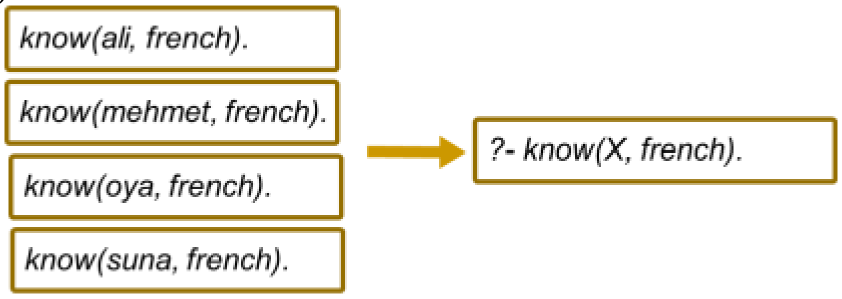
(18) $\exists x$ *know(x, french).*

In general, a query *?- pred*$(X_1, ..., X_n)$ which contains the variables $X_1, ..., X_n$ translates into FOL as:

(19) $\exists x_1, ..., \exists x_n$ *pred*$(X_1, ..., X_n)$.

We can now define our fourth deduction rule, generalization: An existential query is a logical consequence of an instance of it. Therefore, the statement in (17) can be deduced from any of the facts in (11) by the deduction rule of generalization:

(20)



Let us end the discussion of quantifiers in prolog with some remarks.

4.3 Some Remarks on Quantification

First, note that even if it is a proper name in natural language (e.g., French, Ali, etc.), to repeat, an atom has to be either a lowercase letter or a string starting with a lowercase letter in prolog.

Any uppercase letter or a string starting with an uppercase letter can only be treated as variables ranging over individuals. More importantly, predicates cannot be variables in prolog. Any use of a variable in a predicate position will result in a syntax error.

Second, those terms which are free of any variables are called **ground** (e.g., *person(mehmet)*); and those containing occurences of variables are called **nonground** (e.g., *person(X)*). The system predicate *ground/1* can be used to check if a term is free of variables or not:

```
(20) ?- ground(person(mehmet)).
      true.

     ?- ground(person(X)).
     false.

     ?- ground(mehmet).
     true.

     ?- ground(X).
     false.
```

Third, like all logic programs Prolog provides almost a complete abstraction from the hardware of the computer. Variables, for instance, do not stand for a location in physical memory, as they do in conventional programming languages (e.g., C/C++, Java). Prolog variables stand for unspecified but single entities that are constituents of the situation described by the program.

5 An Epistemological Comparison of Propositional Logic and First-Order Logic

What kind of entities are propositions? This is not a question easy to answer. However, even if we cannot provide a full-fledged answer for it, we can approximate the answer.

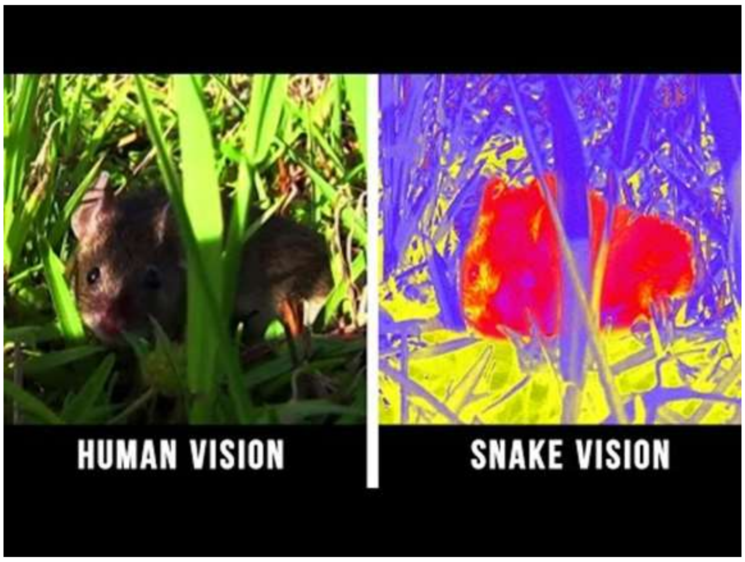
The first step in this approximation is to observe that propositions have existence in mind. That is without any minds there would be no propositions. Hence, it seems reasonable to take propositions as contents of **mental state**s.

Furthermore, we also know that propositions are entities that are either true or false. Somewhat in line with the *correspondence theory* of truth, we are using "true" to indicate that whatever a proposition is, it is something like "corresponding to a situation outside there" (and "false" to make an indication in the other way around). Hence, it is not unreasonable to consider **external situation**s as the contents of propositions.

Propositions can provide mental states with contents at different **epistemological level**s depending on the logic systems they belong to. Atomic (and unanalyzed) propositions of propositional logic come into being at the **existential** epistemological level: they serve to state the existence of a situation but they do not describe the internal structure of the situation. However, propositions of first-order logic come into being at the **predicational** epistemological level: they describe situations by ascribing their constituents some properties.

Interestingly, the above-mentioned contrast between the logic systems parallels the contrast between human and reptile vision:

(21)



Cf. https://in.pinterest.com/pin/snakes-see-the-world-like-this--837880705656191268/

Of course, there is much to study concerning this topic.

6 Conclusion

The main points we have made in this lecture are the following.

A programming language comes with data and control structures. While the logical term is the single data structure in logic programming, control amounts to logical reasoning in this programming paradigm.

A logical term can be either compound or non-compound. A compound term can be either a constant, which must be a lowercase letter or a string starting with a lowercase letter; or a variable, which must be an uppercase letter or an underscore or a string starting with an uppercase letter or an underscore. A compound term is a recursively defined predicate-argument structure. The recursive character stems from the fact that an argument of that structure can be another compound term itself.

If propositions in facts, rules, and queries are structured as compound terms where arguments are allowed to be variables, we get to a first-order-logic-based logic-programming environment.

Variables in a logic programs are assumed to be universally quantified whereas those in query are taken to be existentially quantified.

7 EXERCISES

A. **What happens if the arity of a functor is 0?**

B. **What happens if the arity of a relation is 1?**

C. **Why is the form defined for compound terms in figure 1 recursive?**

D. **Can you explain why predicates must start with lowercase letters?**

Reference Texts

Sterling, L. and Shapiro E. (1986) The Art of prolog. Cambridge, Massachusetts: MIT Press.