Imperial College London

Department of Computing

Third Year Individual Project

LOST: The LOgic Semantics Tutor

Author:
Alina Boghiu

Supervisor:
Ian Hodkinson

Second marker: Fariba Sadri

February 6, 2013

1 Introduction

First order logic is a powerful tool, of great importance in computing and mathematics. For students it is a way of practicing and developing important logical skills and it provides a formal mean for studying and understanding common mathematical structures. However it is difficult for most students to learn the semantics of first order logic and with this being the key to assigning a truth value to any first-order logical sentence, the issue is important.

On the current market, there are a considerable number of tools designed to help teach natural deduction or equivalances, but very few attempts have been made to design something that will acompany the student in learning the semantics of first order logic. Feedback is currently restricted to lectures and tutorials which means this kind of instant guidance provided by computer software would make a great impact in the way students learn, improving both their and the teacher's experience. Designing such a tool raises interesting questions of what would make an interface intuitive and what does it need to provide the user with, such that they can learn and experiment with the semantics of first order predicate logic. From the actual content point of view it would be useful if the user could visualize structures in such a way that they could easily guess the meaning of a possible sentence within it. They should be able to use a toolbox-like signature to add and remove objects and relations as well as have a way of introducing and evaluating sentences. From the software development poit of view arise questions of platform compatibility, graphic interface packages, database handling, storing format etc. Overall it makes for an interesting task with numerous possible extensions.

The idea of a LOgic Semantics Tool (LOST) might sound simple but it is not easy to implement in an attractive way. The risk of making yet another slightly interactive tutorial is quite high therefore my aproach was to make sure this tool engages the student fully, whilst somehow motivating them to try and practice with it as much as they would with any computer game. This shaped the main idea for tackling the problem into designing a computer game where the user is assigned an avatar of his choice which they then have to train into mastering first order logic semantics. Their treinee can then complete small to complex logical missions like recognising the validity of a sentence or playing Hintikka game for which they will be awarded points. In a successful implementation a highscore database will be maintained. Finally, aquiring these points could be set as an assignment by lecturers and other further uses could be implemented.

The main purpose of this tool remains the most important. A minimum viable program, which allows the user to write and play around with structures, would still be a great addition to any logic student's tools. In the following sections I will explain exactly what the software wants to achieve, what stage it is at and the plan and timetable I will follow towards my final implementation.

2 Background

2.1 First order predicate logic and its semantics

In order to understand the product I am aiming for we should first take a look at what first order predicate logic is and why its semantics can be tricky. As an extension of propositional logic, it expreses statements like *Socrates is a man* in much more detail. While propositional logic would regard such a sentence as atomic and assign it a truth value, predicate logic provides a way of describing its internal structure and evaluating it inside a context. It does this by introducing:

- *Constants* - which name the objects (or terms) inside a context (or structure) (e.g. Socrates)
- *Relations* - which describe properties of the objects they take as arguements or just general properties of the structure if they are nullary (i.e. take no arguements) (e.g *man* is a unary relation)

.
These elements form the signature of a structure. With these concepts we can now rewrite the sentence as *man(Socrates)* however we still need a context in order to interpret it and decide its truth value. This is called a structure. We must ask two questions: Which is the object that Socrates describes and what does it mean for something to be a man. If we take our structure to be an imaginary world of dwarfs and name one of them Socrates, our sentence would be false. However in the context of the real world where everyone is human and Socrates refers to the famous philosopher, the sentence is true.

Next, if we want to express *All men are mortal* we must introduce the two quantifiers:

- *(Exists)* - which checks that there is at least one object in the structure to make the sentence it refers to true.
- *(For all)* - which checks that all of the objects in the structure satisfy the sentence it refers to.

.
Now the sentence can now be written as $\forall(men(x) \rightarrow mortal(x))$ and we refer to x as a variable which in this case is bound by the for all quantifier. Another aspect that the user must understand is that sentences that contain unbound variables are valid but cannot be evaluated to a truth value.

2.1.1 The LOST application

General description

The main purpose of LOST will be to model a visual representation of a structure that is intuitive to understand and easy to manipulate. Below is a picture of roughtly how the example discussed would look like. However this is only a prototype at this stage.

The main window (the top left area) contains the structure built with elements from the signature on the right handside. At the bottom the user can input sentences for evaluation. Finally, on the right of the avatar there is a summary of the users activity.

In order to evaluate the sentence *man(Socrates)* you should try and find the object described by Socrates and verify whether the unary relation *man* applies to it. Objects will be represented by circles filled with different colours, according to which unary relations apply to it. If a constant describes an object then that object will also contain the constant's name. Looking at the signature you can see the relation *man* is red. This means all the objects which it applies to will contain this colour. So it should now then easy to see that wheather our sentence is true in this context, just by finding a blue circle labeled Socrates. Finally, The binary relations will be represented by arrows from the first to the second arguement of the relation.
The User should be able to modify the structure as well as the signature. The *Add New Object* button will add new circles, by default black to represent that they are not ground. The user should also be able to save the structure along with its signature as well as to load predefined ones.

Progress so far

One of the first steps when starting the implementation was choosing the main programming language. I took into consideration several facts: I am aiming for quite a complex user interface, I want to develop a desktop application compatible with most platforms to satisfy the various preferences of computing studens and the limited amount of time suggests I should pick a language I am very comfortable with. For these reasons I decided to use Java on the Netbeans IDE which provides a helpful Swing GUI builder.

1. The parser:
   This was the first tool I worked on. Having the experience of MAlice from the previous academic year,

I decided to use Antlr to generate the parser for first order logic sentences. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. From a grammar, ANTLR generates a parser that can build and walk parse trees. At this point I decided the GUI should provide the user a means of easily entering these sentences which would then be passed through to the parser in the form of a string object. I then proceeded to writing the grammar. Another choice was to use the latest version of Antlr, version 4, for its new adaptive parsing strategy which allowes a more relaxed grammar (it accepts even left recursive grammars, except for indirectly left recursive grammars where x calls y which calls x). When writing the grammar I made sure the presidence of operators is taken into account when bulding the parse tree. The Parser also checks that brackets are correctly placed and overall leaves no room for ambiguity.

2. The Logic Tree structure
   The parse tree is not a neat way of storing the input sentence. It is hard to read and evaluate. Therefore I decided to build my own structure, purposely built for logic trees, in order to make the evaluation process clear and also provide a way of displaying nice trees to the user at their request.

   The main class is LogicTree. Here is a description of the most important elements inside it:

```
public class LogicTree {

    LogicTreeNode head;
    Signature signatureBuilder;  \\Keeps track of all the objects and relations that
                                   occur during the build of the logic tree and
                                   stores them such that a relevant signature can be
                                   automatically generated for the input sentence.

    public LogicTree(ParseTree c) {  \\Builds the logic tree from the parse tree
                                        generated by the parser
    ...
    }

    evaluate(Structure s) \\Evaluates the tree representing the input sentence,
                            from the top down, according to the evaluation rules
                            in each node and according to the given structure.
            throws UnboundException {
        return head.evaluate(s);
    }
...
}
```

3. The signature
4. The sentence
5. Sentence evaluation

The firt natural step was to think what the features of the minimum viable product would be. When deciding I took into account the project specifications and available time. Here is an outline of what a basic product should be able to to:

- Provide an interface which allows the user to define signatures, build structures and input logic sentences:

- Evaluate sentences given a structure
- Store structures, signatures and sentences
- Provide a collection of predifined structures, signatures and sentences.

You may realise this is already quite complex to implement, however it provides little functionality. Therefore, if time allowes, I am hoping to further implement the followinf features:

- Implement a hitikka game for the avatar to play against the computer.
- Allow multimple users
-

2.2 First Order Predicate Logic

As a powerful extension of propositional logic, predicate logic is extremely important. Propositional logic is quite nice, but not very expressive. Some statements need something more than propositional logic to express.

Phrases such as *the computer is a Sun* and *Frank bought grapes* are no longer regarded as atomic in predicate logic. Looking at their internal structure we could have: A relation symbol (or predicate symbol) Sun. It takes 1 argument we say it is unary or its arity is 1. We can also introduce a relation symbol bought. It takes 2 arguments we say it is binary, or its arity is 2. Constants, to name objects. Then Sun(Heron) and bought(Frank,grapes) are two new atomic formulas.

2.3 Previous Work

Previous attempts have been made to fill this gap in the market. The first one that came to mind was the solution one student provided for this same specification in 2007. They developed a Java application which meets the minimum requirements and provides a few extrafeatures like the Hintikka game and an attempt at Logic-English translation. This LOST tool allowes users to enter signatures, structures and sentences, then evaluate them. Tarski's World Hyperproof Language, PRoof and Logic

2.4 Hintikka

One good way of understanding first order logic semantics is by playing the Hintikka game. It involves two players which are assigned one of the quantifiers. The then work on a sentence within a signature:

$$\exists$$

tries to make the sentence true while

$$\forall$$

tries to make it false. To better understand how the game works we must look at the tree for the given sentence.

At each point in the game, we look at the node in the tree where the game is currently at. If that node is of the form: $\forall$ x, then the $\forall$ player has to choose a value for x, i.e. an object in the structure $\exists$ x, then the $\exists$ player has to choose a value for x, i.e. an object in the structure , then the $\forall$ player chooses the next branch to follow down the tree , then the $\exists$ player chooses the next branch to follow down the tree , regard AB as (AB) (AB) , regard AB as (AB) , then swap labels $\exists$ and $\forall$. The player who was $\exists$ becomes $\forall$ and vice versa. an atomic (or quantifier-free) formula, then evaluate the formula in the given structure with the current values for the variables: o If the formula is true, the $\exists$ player wins o If it is false, the $\forall$ player wins 25 LOST - Logic Semantics Tutor The game ends when a leaf (atomic formula) is reached. Therefore, during the game each player should try to lead the game towards a leaf that will make him/her win. This will be a leaf which evaluates to true if the player is $\exists$, and a leaf which evaluates to false if the player is $\forall$.

3 Project Plan and Timetable

So far I've described my final product in reasonable detail. In order to achieve this it is important to prioritise the implementation of some features and lay out a plan and timetable for the approach.

3.1 Up to this point

I have managed to wite a comprehensive grammar

- write a first order predicate logic grammar for ANTLR and build a parser
- create a tree structure that can evaluate to a boolean given a signature
- generate tree nodes from the parsed input to fit the LogicTree structure
- generate a signature builder from the parsed input
- create a graphical interface to allow easy input of sentences
- extend the interface to display the current signature
- extend the interface to allow the user to modify the signature
-

4 Evaluation