SHERLOCK — A Neural Network Software for Automated Problem Solving

Ekaterina Komendantskaya[1] and Qiming Zhang[1]

School of Computing, University of Dundee, UK

Abstract. We propose SHERLOCK - a novel problem-solving application based on neuro-symbolic networks. The application takes a knowledge base and rules in the form of a logic program, and compiles it into a connectionist neural network that performs computations. The network's output signal is then translated back into logical form. SHERLOCK allows to compile logic programs either to classical neuro-symbolic networks (the "core method"), or to inductive neural networks (CILP) — the latter can be trained using back-propagation methods.
Key words: Neural Networks and Connectionist Models, Hybrid Neuro-Symbolic Systems, Decision Support Systems, Reasoning Methods, Knowledge Acquisition and Representation.

1 Introduction

Computational logic and Neurocomputing are the two different paradigms that underly numerous attempts to expand and refine the qualities and capacities of AI. *Neuro-Symbolic Integration* [2, 3, 6, 12] is the area of research that endeavors to synthesize the best of the two worlds. The goal of neuro-symbolic computation is to bridge the gap between robust machine learning based on neural networks and expressive forms of reasoning normally implemented by symbolic computation. The neural networks provide the machinery for effective, massively-parallel computation while symbolic reasoning provides a formal language and explanation to the neural model. More generally, neuro-symbolic computation offers a methodology for the creation of sound cognitive models of computation. The Neuro-Symbolism has since developed different approaches to inductive, probabilistic, and fuzzy logic programming; [6, 2, 3, 14].

In this paper, we take the ideas of neuro-symbolic integration to the level of software engineering and design. That is, we do not consider theoretical aspects of neuro-symbolic integration here, but take its synthetic principle to be our main software engineering principle. So, which methods could software engineering borrow from the area of neuro-symbolic integration? In this paper, we offer one possible answer, but see also [1].

Declarative programming languages, and especially logic programming, have one important underlying idea — they are designed to be syntactically similar to the way people reason. Logic programming, for example, is one of the easiest languages to teach students with non-technical background or general public alike. Also, it is feasible to parse natural language into logic programming syntax. Therefore, the strength of logic programming from the software engineering point of view is that it makes for a general and easily accessible interface for users with diverse backgrounds.

Neural networks, on the other hand, offer both massive parallelism and ability to adapt. However, it would seem almost impossible to imagine that a person with non-technical background easily masters neural networks as part of his working routine, alongside with a web-browser or a text editor. It is common that industrial applications of neural networks are designed and maintained by specialists, while non-specialist users do not have ways to edit the applications. This is why neural network applications are often problem-specific. Such applications could be made more general and user-friendly if the users were given a nice easy interface to manipulate neural networks at a level of natural language.

Example 1. Consider a police officer who has just come to a crime scene and wishes to record all evidence available. To be efficient, the police officer uses a small portable computer that has a problem-solving assistant. What should this assistant be like? Neural network software would come in handy, because it can be trained as new evidence is obtained; also – it can be fast due to parallelism. On top of this neural software, though, it is best to have an easy interface allowing the officer to enter data in the form of a natural language.

In this paper, we propose SHERLOCK — an application that allows the user to type in the knowledge base in the language close to the natural language, and then rely on the compiler that transforms the problem into a suitable neural network. The network will attempt to solve the problem; and once the solution is found — it outputs the answer in a logical form. Thus, SHERLOCK successfully implements the full *neuro-symbolic cycle*, [6, 3]. Additionally, as we explain in Section 4, SHERLOCK can be embedded into a bigger knowledge-refining cycle.

Our results relate to the work of [4] proposing a neural compiler for PASCAL; and the programming languages AEL, NETDEF designed to be compiled by neural networks, [13]. SHERLOCK differs from the previous similar work in two respects. It is the first fully automated neural compiler for declarative languages we know of. Also, unlike [4], where the main emphasis was on building a fully functional complier for a programming language; here our emphasis is not on creating a neural compiler for PROLOG *per se*; but building a compiler sufficient to handle knowledge bases and reason over them.

The paper is organised as follows. Section 2 contains background definitions. Section 3 describes the interface and design of SHERLOCK. Section 4 shows the knowledge refinement scheme with SHERLOCK. In Section 5, we conclude.

2 Preliminaries

2.1 Logic programs

In the standard formulations of logic programming, such as in Lloyd's book [11], a first-order logic program P consists of a finite set of clauses of the form

$A \leftarrow L_1, \ldots, L_n$, where A is an atom, and the L_i's are literals — that is, possibly negated atomic formulae, typically containing free variables, and where $L_1, \ldots, L_n$ is understood to mean the conjunction of the L_i's: note that n may be 0. We assume that the reader is familiar with first-order logic, and the notions of first-order terms and atomic formulae, as can be found e.g. in [11].

Example 2. Consider a detective story: A small bank was robbed. The bank safe is found open. Footprints left on the floor belong to a person with a pair of small feet; and burned cigarettes are found on the scene. The detective came up with a deduction rule: "The offender had keys for bank safe, small feet and the habit of smoking". Then the detective found the following facts about the bank staff:
1) "Jane, Harry and Stephen had the keys."
2) "Stephen and Jane have small feet."
3) "Stephen has a habit of smoking."
The detective drew the conclusion logically that Stephen was the criminal.

Here is a logic program which describes exactly this toy scenario:

$$\begin{aligned}
Criminal(X) &\leftarrow HasKeys(X), SmallFeet(X), Smoke(X)\\
HasKeys(Harry) &\leftarrow\\
HasKeys(Jane) &\leftarrow\\
HasKeys(Stephen) &\leftarrow\\
SmallFeet(Stephen) &\leftarrow\\
SmallFeet(Jane) &\leftarrow\\
Smoke(Stephen) &\leftarrow
\end{aligned}$$

Models satisfying Logic programs can be characterised by the fixed-point semantic operators; the one most commonly used is the T_P-operator; see also [11]. For the lack of space, we will not go into much of detail here, but we will use the theorems relating the fixed point-operator T_P and neural networks in the latter sections; see also [8, 9, 2].

Example 3. Considering the logic program from Example 2: $T_P \uparrow 0 = \emptyset$;
$T_P \uparrow 1$ = { HasKeys(Harry), HasKeys(Jane), HasKeys(Stephen), SmallFeet(Stephen), SmallFeet(Jane),Smoke(Stephen) };
$T_P \uparrow 2$ = { HasKeys(Harry), HasKeys(Jane), HasKeys(Stephen), SmallFeet(Stephen), SmallFeet(Jane),Smoke(Stephen), Criminal(Stephen) }
$T_P \uparrow 2 = T_P \uparrow 3 = lfp(T_P)$.

2.2 Neuro-Symbolic Networks

Here, we briefly describe two neuro-symbolic networks we use when designing SHERLOCK. We assume the reader is familiar with basic definitions of neural networks, as e.g. given in [7]. The first kind of networks we consider here are called T_P*-neural networks* for their correspondence to the semantic operator T_P

[11], and were originally proposed in [8, 9]. The second kind is the *Connectionist Inductive Learning and Logic Programming (CILP)* [2] — a massively parallel computational model based on a feed-forward artificial neural network that integrates inductive learning from examples and background knowledge with deductive reasoning using logic programming.

Algorithm for constructing T_P-neural networks.
Given a logic program P, we take all the ground instances of clauses in P — that is, we make all possible substitutions of variable-free terms for variables in P. Let p and q be the number of literals and the number of clauses occurring in P, respectively. Without loss of generality, we may assume that the ground atoms are numbered from 1 to p. The network N associated with P can now be constructed as follows:

1. The input and output layer is a vector of p binary threshold neurons, where the i-th neuron represents the atom A_i, $1 \leq i \leq p$. The threshold of each neuron occurring in the input or output layer is set to 0.5.
2. For each clause of the form $A \leftarrow A_1, \ldots, A_m, \sim A_{m+1}, \ldots, \sim A_n (0 \leq m \leq n)$,occurring in P do the following:
3. Add a binary threshold unit c to the hidden layer of N.
4. Connect c to the unit representing A in the output layer with weight 1.
5. For each A_i, $1 \leq i \leq m$, connect the unit representing A_i in the input layer of N to c, and set the connection weight to 1. For each A_i, $m + 1 \leq i \leq n$, connect the unit representing A_i in the input layer of N to c, and set the connection weight to -1. Set the threshold θ_c of c to $m - 0.5$.

Theorem 1. *There exists a single hidden layer recurrent network such that each computation starting with an arbitrary initial input I converges to a stable state and yields the unique fixpoint of T_P.*

Here is the algorithm to compute the least Herbrand Model of P in a T_P-neural network.

Massively Parallel Deduction Algorithm for T_P-networks.
Let p be the number of input neurons and the number of output neurons in T_P-neural network N. The input is defined by a vector $I = (I_1, \ldots, I_p)$ and the output is given by a vector $O = (O_1, \ldots, O_p)$.

```
1. Initialize $I = [0, 0, \ldots, 0]$.
2. Loop:
    – Calculate $O$ = feed-forward($I$);
    – If $I$ is equal to $O$, then terminate;
    – If $I$ is not equal to $O$, then for $j$ in $(1, \ldots, p)$, replace the value of $I_j$
      with the value of $O_j$ in $O$.
```

Example 4. Consider the logic program from Example 2. The network N for it will be recurrently connected, its output vector feeds the input vector at every iteration of T_P. Let the initial input vector $I = (0, 0, 0, 0, 0)$. So the deduction process would be as follows:

$T_P \uparrow 0 = [0, 0, 0, 0, 0]$;
$T_P \uparrow 1 = [0, 1, 0, 0, 0]$;
$T_P \uparrow 2 = [1, 1, 0, 0, 0]$;
$T_P \uparrow 2 = T_P \uparrow 3 = lfp(T_P)$.

To embed the propositional knowledge of a general logic program P in a neural network N, CILP [2] uses an approach similar to T_P-Neural Networks. However, a CILP-neural network N deploys a semi-linear function as its activation function:

$$f(x) = \frac{2}{(1 + e^{-\beta x})} - 1.$$

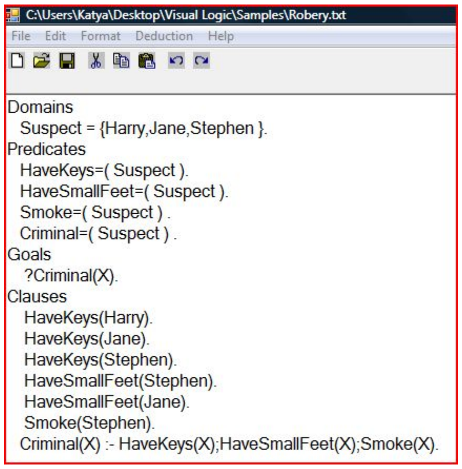
The function $f(x)$ has the real numbers as domain and $[-1, 1]$ as codomain.

For each propositional general program P, there exists a feed-forward artificial neural network N (CILP) with one hidden layer and semi-linear neurons such that N computes T_P, see e.g. [2] for more details.

3 Design of SHERLOCK

The neural-symbolic systems can do deduction. However, it is very difficult for users to use such systems because they would have to embed the symbolic knowledge into a neural network by setting up weights and thresholds and then interpret the outcome of the neural network. This is why, a convenient interface between users and neural-symbolic systems is required, and we propose one such interface in this section.

Example 5. The Figure 1 shows SHERLOCK's interface together with the logic program from Example 2.


C:\Users\Katya\Desktop\Visual Logic\Samples\Robery.txt
File Edit Format Deduction Help

```
Domains
  Suspect = {Harry,Jane,Stephen }.
Predicates
  HaveKeys=( Suspect ).
  HaveSmallFeet=( Suspect ).
  Smoke=( Suspect ) .
  Criminal=( Suspect ) .
Goals
   ?Criminal(X).
Clauses
   HaveKeys(Harry).
   HaveKeys(Jane).
   HaveKeys(Stephen).
   HaveSmallFeet(Stephen).
   HaveSmallFeet(Jane).
   Smoke(Stephen).
  Criminal(X) :- HaveKeys(X);HaveSmallFeet(X);Smoke(X).
```

Fig. 1. SHERLOCK's interface together with Logic program from Example 2

SHERLOCK consist of the following components:

1. A code editor, in which the users can write a general logic program in a prolog-like declarative language;
2. A translator, which can analyse syntax and semantics of the logic program to set up neural-symbolic systems according to the logic program;
3. A model of T_P-neural networks, and a model of CILP-neural networks;
4. An interpreter;
5. An output reader.

First, we implement the **code editor**, see also Figure 1. The code editor for SHERLOCK is a simple text editor, in which users write logic programs. Once the data is entered, it allows the user to choose to which neuro-symbolic network the problem will be compiled by using the "Deduction" button from the drop-out menu; and then choosing "T_P-network" or "CILP network".

A logic program consists of facts, rules and questions. In order to represent these, the code editor for SHERLOCK is designed to have four sections `Domains`, `Predicates`, `Goals`, and `Rules`.

We design a simple grammar by defining the following language L as follows:

– `Domains` format: $<$ domain name $>= \{<$ list of terms $>\}$.
– `Predicate` format: $<$ predicate name $>= (<$ list of domain $>)$.
– `Goal` format: ? $<$ predicate name $>$ $(<$ list of terms or variables $>)$.
`Rules` divide into facts and clauses.
– Facts format: $<$ predicate name $>$ $(<$ list of terms $>)$.
– Clauses format: $<$ predicate name $>$ $(<$ list of terms or variables $>)$ $: -$ $<$ predicate name $>$ $(<$ list of terms or variables $>)\{<$ predicate name $>$ $(<$ list of terms or variables $>)\}$.

Example 6. For an example how this grammar is applied in SHERLOCK, see Figure 1. We propose yet another example here. We continue the scenario of Example 2. After concluding that Stephen was the offender, the detective notices that Stephen's feet are bigger than the footprints left at the crime scene. He needs to find out who might have smaller feet than Stephen's. For this, another piece of data could be added:

```
Domains
Suspect={Harry, Jane, Stephen}.
Predicates
smaller=(Suspect, Suspect).
Goals
? smaller(X, Stephen).
Clauses
smaller(Jane, Hurry).
smaller(Hurry, Stephen).
smaller(X,Y):-smaller(X,Z); smaller(Z,Y).
```

For the lack of space, we use a simplified version of what could be the real situation. If the real-life database is sufficiently big, one might well use SHERLOCK to handle it. In real-life situation, the program may contain feet sizes for each staff member, here we included "`smaller`" instead.

Next, we define **models** for neuro-symbolic networks.

Definition 1. *Given a logic program P,* a Model for a T_P-neural network *is a tuple (n, q, W_1, W_2), with q being the number of clauses, n being the number of all propositional variables, W_1 an n-by-q matrix describing weights between the input layer and the hidden layer; and W_2 a q-by-n matrix describing weights between the hidden layer and the output layer. In the output layer, there are k thresholds, whose values are set* 0.5*. In the hidden layer there are q thresholds, whose values are determined by W_1. The q thresholds θ_l ($l \in [1, \ldots, q]$) in the hidden layer are calculated as follows: for $l \in [1, \ldots, q]$, let p_l be the positive value of the l-th column of W_1. Then*

$$\theta_l = p_l - 0.5$$

Similarly, we define a model for the CILP-neural network; modulo the changes in the transfer functions. The model of the CILP-neural network will also be defined by a tuple (n, q, W_1, W_2).

A **translator** is a necessary component which connects the symbolic logic programming component to the connectionist computing systems. It transforms a logic program written in the language L into a tuple $(n, q, W1, W2)$ which is either the model of a T_P-neural network or the model of a CILP-neural network — depending on the user's choice.

The translator is very similar to a single pass compiler which consists of two stages. In the first stage, the translator performs *lexical analysis*, *syntactic analysis* and creates a *token table*. In the second stage, it performs *intermediate representation* generation and creates the *target tuple* (n, q, W_1, W_2).

Syntax analysis is a phase in which the overall structure of a program is identified, and involves an understanding of the order in which the symbols in a program may appear. According to the declarative language L, there are four sections in the problem specification given to SHERLOCK - Domains, Predicates and Goals. Each section has a label to mark it and there is a piece of syntax which is used to define its contents, see Appendix A.

According to these grammars, the syntax analyser (or parser) can fit a sequence of tokens into a specified syntax. A parsing problem consists of finding a derivation (if one exists) of a particular sentence using the given grammar. In this translator, the parser is a left to right bottom-up parser with one symbol lookahead LR (1). The LR (1) is a very common algorithm in complier design, which will not be introduced here, see [5, 10].

Intermediate representation generation is a phase to transform each statement in the original code of a logic program to intermediate data according to a particular syntax.

Given a logic program P, the translator generates appropriate intermediate data which it can use to create *the target tuple* to construct a neural-symbolic network representing P.

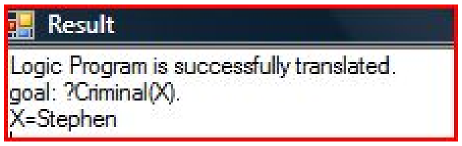
Target tuple creation is a simple phase to create a model tuple (n, q, W_1, W_2) according to the syntax, see also Section 2.2. The neural-symbolic system could be either a T_P-neural network or a CILP neural network, which depends on the choice of users.

Interpreter and Output reader.
An **interpreter** is a component that interprets the execution result of a neural-symbolic system into truth values and gives symbolic answers to the goals in P.

According to the record of the model of the neurons in the output layer, it is easy to judge a propositional variable truth value. For a T_P-neural network, if the value of a neuron is greater than or equal to 0.5, then its corresponding propositional variable is assigned *true*; otherwise the propositional variable is assigned *false*. For a CILP- neural network, if the value of a neuron is greater than or equal to A_{min}, then its corresponding propositional variable is assigned *true*.

The interpreter "understands" the meaning of goals in a logic program as follows. It searches for answers and then formats answers in a symbolic way. There are two types of goals: one has no variables and the other does. A goal with no variables requires an answer to be the truth value. A goal with variables requires an answer that presents the sets of substitutions that make the goal true. SHERLOCK allows multiple goals and relational goals, see Appendix B.

Example 7. Figure 2 shows the answer to the logic program from Figure 1.



```
Logic Program is successfully translated.
goal: ?Criminal(X).
X=Stephen
```

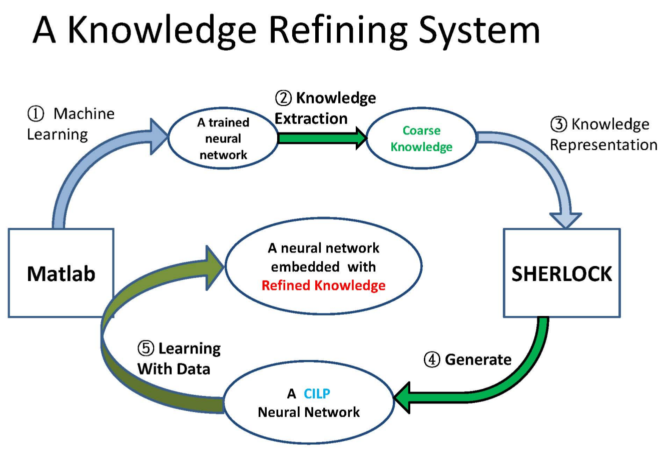
Fig. 2. SHERLOCK's answer to the goal `Criminal(X)` for the logic program from Example 2.

In case of multiple goals and multiple possible answers, SHERLOCK outputs all answers that satisfy the model specification, see Figure 3. This feature is useful in crime investigation applications, as it may help to form new hypotheses.

4 Knowledge Refining using SHERLOCK

Knowledge refining is one of the important feature in human reasoning, and crime investigation is not an exception. We wish to insert background (or "coarse") knowledge into a neural network and obtain refined knowledge by learning with example data. One kind of Neural-symbolic systems CILP is very suitable to do knowledge refining. Not only CILP has the capability to present background knowledge into neural networks, but also it can use back-propagation to get networks trained with examples. We propose a novel approach to build knowledge refining systems based on SHERLOCK:

1. Coarse knowledge is obtained from the trained neural network using one of the standard extraction techniques.
2. Then it is expressed in the first order language in SHERLOCK.
3. A CILP neural network is obtained.
4. CILP is trained with the data, and the embedded knowledge is refined.



We test this model on the famous cancer data set from the UCI Machine Learning Repository. The final neural network has a performance of 96.7%. The performance of the final neural network cannot be improved by setting a better training goal while a general neural network can. This implies the knowledge embedded in the CILP neural network is sensitive to certain kinds of data.

We summarise the properties of this model as follows:

1. It provides a methodology to obtain knowledge in any domain by using both induction and deduction.
2. If the knowledge obtained in Step 1 is reasonable, the final neural network will remain a clear structure, which could be interpreted to symbolic knowledge. Otherwise, the neural network is just an ordinary supervised trained neural network.
3. The final neural network has a very good performance in terms of learning. Besides, it seems that the neural network owns an ability to detect some fault data due to the knowledge embedded in it.

5 Evaluation, Conclusions and Future Work

We successfully tested SHERLOCK on several more challenging examples, notably on the famous Einstein's riddle known for its hardness, and cancer data sets. For more tested examples and SHERLOCK software see [15].

SHERLOCK performs the *full neuro-symbolic cycle* — starting from the knowledge base in its logical form, then translating it into a chosen kind of neural networks and then interpreting the output of the neural network in a logical form. It can additionally be embedded into a bigger machine-learning cycle as explained in Section 4.

We see SHERLOCK's future as a handy application for researchers in the field of Neuro-Symbolic Integration [2, 3, 6] or data mining, but also in real-life situations, where fast parallel computations, knowledge revision and training from data are important components of the information management. We have presented one such application here in the area of crime detection, but the design and interface of SHERLOCK accepts a very general language, and can have a wide range of applications.

References

1. I. Cloete and J. M. Zurada. *Knowledge-Based Neurocomputing.* MIT Press, 2000.
2. A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications.* Springer-Verlag, 2002.
3. A. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay. *Neural-Symbolic Cognitive Reasoning.* Cognitive Technologies. Springer-Verlag, 2008.
4. F. Gruau, J.-Y. Ratajszczak, and G. Wiber. A neural compiler. *Theor. Comput. Sci.*, 141(1&2):1–52, 1995.
5. D. Grune, C. Jacobs, and K. Langendoen. *Modern Compiler Design.* John Wiley, New York, 2000.
6. B. Hammer and P. Hitzler. *Perspectives of Neural-Symbolic Integration.* Studies in Computational Intelligence. Springer Verlag, 2007.
7. S. Haykin. *Neural Networks. A Comprehensive Foundation.* Macmillan College Publishing Company, 1994.
8. S. Hölldobler and Y. Kalinke. Towards a massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.
9. S. Hölldobler, Y. Kalinke, and H. P. Storr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.
10. R. Hunter. *The Essense of Compilers.* London: Prentice Hall, 1999.
11. J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
12. J. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data.* Springer, Cognitive Technologies Series, 2003.
13. H. Siegelmann. Neural programming language. *Conference of the American Association for Artificial Intelligence*, 1994.
14. J. Wang and P. Domingos. Hybrid markov logic networks. In *AAAI*, pages 1106–1111, 2008.
15. Q. Zhang. Sherlock: neuro-symbolic reasoner. Software and experiments, 2011. www.computing.dundee.ac.uk/staff/katya/sherlock.

A Syntax analysis in SHERLOCK

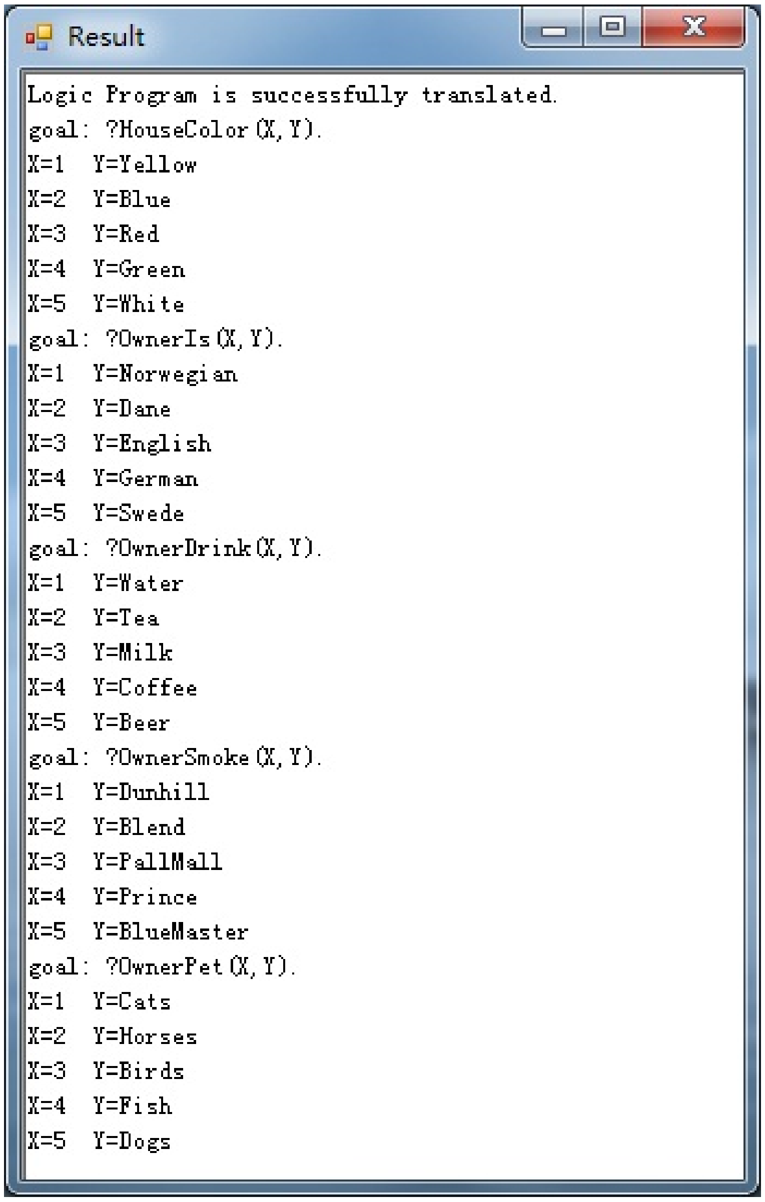
Example 8. We include some parts of the syntax analysis included in the SHERLOCK designed below.

- Labels/Key words: `Domains`, `Predicates`, `Goals`, `Rules`. Its syntax is defined as follows:
  S $\rightarrow$ `Domains` | `Predicates` | `Goals` | `Rules`

- `Domains:`
  S $\rightarrow$ DOMAIN = $\{B\}$
  B $\rightarrow$ TERM
  T $\rightarrow$ TERM, B
  DOMAIN $\rightarrow$ [a-zA-Z]
  DOMAIN $\rightarrow$ [a-zA-Z]DOMAIN
  TERM $\rightarrow$ [a-z] [a-zA-Z]*

- `Predicates`
  S $\rightarrow$ PREDICATE = (B) .
  B $\rightarrow$ DOMAIN
  B $\rightarrow$ DOMAIN, B
  PREDICATE $\rightarrow$ [a-zA-Z] [a-zA-Z]*
  DOMAIN $\rightarrow$ [a-zA-Z] [a-zA-Z]*
- `Goals`
  $\vdots$

B Answer to Einstein's riddle



Result

```
Logic Program is successfully translated.
goal: ?HouseColor(X,Y).
X=1  Y=Yellow
X=2  Y=Blue
X=3  Y=Red
X=4  Y=Green
X=5  Y=White
goal: ?OwnerIs(X,Y).
X=1  Y=Norwegian
X=2  Y=Dane
X=3  Y=English
X=4  Y=German
X=5  Y=Swede
goal: ?OwnerDrink(X,Y).
X=1  Y=Water
X=2  Y=Tea
X=3  Y=Milk
X=4  Y=Coffee
X=5  Y=Beer
goal: ?OwnerSmoke(X,Y).
X=1  Y=Dunhill
X=2  Y=Blend
X=3  Y=PallMall
X=4  Y=Prince
X=5  Y=BlueMaster
goal: ?OwnerPet(X,Y).
X=1  Y=Cats
X=2  Y=Horses
X=3  Y=Birds
X=4  Y=Fish
X=5  Y=Dogs
```

Fig. 3. SHERLOCK's answer to Einstein's riddle.