CS301 Final Project

CS301

Jun Zhou

Professor Tao

**Introduction**

Prolog stands for Programming in logic. It is a declarative programming language that based on first-order logic. Prolog is a homoiconic language, which means Prolog programs are also valid Prolog terms.

Prolog is most frequently used in these following areas:

1. Natural-Language Processing

Prolog is one of the first language for this topic. Prolog language models can take natural languages and apply a knowledge base of facets can inferences on it. Then it can translate that complex language into concrete rules that can be understood by the computer.

1. Semantic Web
2. Games

Prolog models can easily express the behaviors of other characters in the system and build different behaviors into different types of entity in the game.

1. Artificial Intelligence

One of the most important area in AI is to build knowledge into machine. This knowledge can be in different forms and the agent have to modifies its behavior based on complex rules. Prolog can build these rules based on formal logic.

1. Scheduling

Prolog can work with constrained resources efficiently and easily.

**Methodology**

The language that I will be learning is prolog.

My study plan for this language is as following:

Step1: Understand the basic information of prolog. What are the advantage and disadvantage of prolog?

Step2: Learn the syntax of prolog

Step3: Use prolog to solve some classic problems

Step4: Understand the basic design of prolog

Prolog is known to be Turing-complete, which means it is as powerful as any other languages. However, no one would use prolog to write a strategic game because Prolog is not a general-purpose language and only well-suit for specific tasks that benefit from rule-based logical queries ([3], pp 133). So, the programs that I chose for this project is the funclang.pl, which is an interpreter for funclang. It is inspired by <https://www.metalevel.at/lisprolog/>.

This program will serve the purpose of this project because it is a complicated prolog program that really pushed the limit of what swi-prolog can do. It utilizes many advanced ideas in prolog programming, including recursion, DCG, backtracking with cut, association list, high order procedure and logical purity. Also, it ties up to what we learn in the class about interpreter implementation.

**The Basics of Prolog**

The syntax of prolog is rather simple. We can declare some facts which builds the knowledge base of the system. Then we can query the knowledge base. A knowledge base can be considered similar to database.

Facts are constituted by assertion (ABox) and relation (TBox). The ABox contains extensional knowledge about the domain of interest ([2], pp.19) for example, John is a student.

Another key element of a DL knowledge base is given by the operations used to build the terminology. Such operations are directly related to the forms and the meaning of the declarations allowed in the TBox. The basic form of declaration in a TBox is a concept *de nition*, that is, the deﬁnition of a new concept in terms of other previously deﬁned concepts ([2], pp17). For example, all students are persons.

When programming in prolog, one doesn’t have to write any algorithm to the problem. Prolog lets programmer express the logic in facts and inferences and then we can ask questions. Programmer is not responsible for building any step-by-step algorithm for this language. Prolog is about describing your world as it is and presenting logical problems that computer can try to solve it ([3], pp103)

The prolog interpreter uses backward chaining algorithm to solve the problem with the rules given. Backward chaining starts with the goal and work backwards attempting to decompose it into a set of clauses that are in the knowledge base. The implementation in prolog of backward chaining is using depth first search on the knowledge base. It will find a complete sequence of propositions for the first subgoal before working on the others.

Prolog supports backtracking and unification as built-in features. Unification is to find the values that make both sides match (([3], pp 104)). Backtracking: trace backward in the decision tree when a branch fail.

all data are represented by Prolog terms. Each term is either a variable, an atomic term or a compound term:

variables start with an uppercase letter or with an underscore (\_). A single underscore denotes an anonymous variable and can be read as "any term". For example, X, Y, \_爱 and Prolog are variables.

atomic terms are: atoms, integers or floating point numbers.

compound terms are defined inductively as follows: If T1, T2, ..., TN are terms, then F(T1, T2, ..., TN) is also a term, where F is called a functor name and adheres to the same syntax rules as atoms. F/N is called the principal functor of the compound term, and N is called the arity. Examples: f(a), g(f(X)) and +(a, f(X)). ([1], pp25)

**Advanced Programming in Prolog**

**1: DCG**

Similar to context free grammar, a Prolog definite clause grammar (DCG) describes a [list](https://www.metalevel.at/prolog/data#list). Operationally, DCGs can be used to parse, generate and check.  
  
A DCG is defined by rules. A DCG rule has the form:

Head --> Body.

A rule's body consists of terminals and nonterminals. A terminal is a list, which stands for the elements it contains. A nonterminal refers to a DCG or other grammar construct, which stand for the elements they themselves describe. We use the nonterminal indicator f//N to refer to the nonterminal f with arity N. Note that // distinguishes it from a Prolog [predicate indicator](https://www.metalevel.at/prolog/concepts#predicate). ([1], pp124)  
  
As an example of a DCG, let us describe lists whose every element is the atom a. We shall use the nonterminal as//0 to refer to such lists:

as --> [].

as --> [a], as.   
  
To invoke a grammar rule, we use Prolog's built-in phrase/2 predicate. The first argument is, syntactically, a DCG body. phrase(Body, Ls) is true iff Body describes the list Ls.([4])  
  
For example, let us use the single nonterminal as//0 to ask for all lists that it describes, by posting the [most general query](https://www.metalevel.at/prolog/concepts#toplevel):

?- phrase(as, Ls).

**Ls = [] ;**

**Ls = [a] ;**

**Ls = [a, a] ;**

**Ls = [a, a, a] ;**

**Ls = [a, a, a, a] ;**

etc.

Examples of more specific queries and the system's answers:

?- phrase(as, [a,a,a]).

**true.**

?- phrase(as, [b,c,d]).

**false.**

?- phrase(as, [a,X,a]).

**X = a.**

**2: Higher-order Predicates**

Call/N:

The call/N family of built-in predicates allow us to call Prolog closures dynamically. A closure is a term that denotes a Prolog goal to which zero or more arguments are added before it is called. The mechanism to invoke arbitrary Prolog goals dynamically is called meta-call, and it is the basic building block by which we can define arbitrary meta-predicates in Prolog. Importantly, the call/N family of predicates retain logical purity of the predicates they call. ([1],pp143)

maplist/N:

The predicates maplist/2 and maplist/3 are among the most important and most frequently used meta-predicates. All widely used Prolog systems provide them as built-in or library predicates. ([1],pp243) Using goal expansion, the meta-call can be compiled away in many cases to improve performance. The primary advantage of maplist/N is that you can lift any relation that holds for a single element (or single pair, triple, etc. of elements) to lists of such elements (or corresponding elements).

The goal maplist(Pred\_1, Ls) is true iff call(Pred\_1, L) is true for each element L in the list Ls.

The goal maplist(Pred\_2, As, Bs) is true iff call(Pred\_2, A, B) is true for each pair of elements A∈As and B∈Bs that have the same index. ([4]). Examples can be found in the funclang.pl.

fold/N:

The fold/N family of predicates describe a fold from the left of a list. You can think of a fold as a list traversal with intermediate states. An intermediate state becomes final when no more elements remain.

The most frequently used of these predicates is fold/4:

fold(Pred\_3, Ls, S0, S) describes a fold from the left of the list Ls, where S0 is the initial state and S is the final state. For each element L∈Ls and intermediate state Sn, call(Pred\_3, L, Sn, Sn+1) is invoked to relate the current list element L and intermediate state Sn to the next intermediate state Sn+1.

Again, these predicates let us focus on the relation for one element at a time, and then lift this relation to lists of elements.

In funclang.pl, fold is implemented as arrith/4. See the following adopted from funcang,pl

/\*arith(a,b,c,d) deals with the multiple inputs of arithmetic expressions

a is the input expressions, b is the operation, c is the current value, d is the value of all expressions

example: (+ 3 4 5):

arith([3 4 5], +, 0, v) :- arith([4 5], +, 3, v) :- arith([5], +, 7, v) :- arith([], +, 12, v):- arith([], +, 12, 12)

\*/

**Logical Purity**

Logical purity refers that program behaves like a relation. There are two general ways to implement the logical purity.

1. *intrinsic definition*, referring to building blocks that guarantee purity by construction
2. *extrinsic definition*, referring to properties that can be tested without knowing details about the code.

In the following, we discuss these variants in more detail.

**Intrinsic definition**:

That is to give an *inductive* definition of predicates that are *pure*.   
  
For example:

* true/0 and false/0 are pure
* (=)/2 and dif/2 are pure
* arithmetic constraints like (#=)/2 and (#\=)/2 are pure
* call(Goal) is pure *iff* Goal is pure
* maplist(Goal, Ls) is pure *iff* Goal is pure
* (A,B) and (A;B) are pure *iff* A and B are pure
* *.........*

For a large class of Prolog programs, this approach makes it very easy to determine whether a specific program is pure. That is a clear advantage of the approach. On the other hand, a drawback of this approach is that it does not explain *why* these predicates are pure, and how we can decide which other predicates, if any, are also pure. (adopted from [5])

**Extrinsic definition**:

A different approach to characterize logical purity only takes into account program properties that can be observed from the outside, without knowing any implementation details.   
  
For example:

* if a goal produces **output** on the terminal, it is *not* pure
* if a goal modifies **global state**, it is *not* pure
* if, for a goal G that contains the subterm S, there is a term T such that (G,S=T) *succeeds unconditionally*, but (S=T,G) *fails*, then G is not **monotonic** and hence *not* pure
* analogously: if, for a goal G that contains the subterm S, there is a term T such that (G,S=T) *fails*, but (S=T,G) *succeeds unconditionally*, then G is *not* **steadfast** and hence *not* pure
* *more cases omitted*

An advantage of this approach is that we get a clearer idea about the *intention* behind logical purity. With this term, we want to denote programs that satisfy laws and invariants that we know from classical first-order logic. Examples of such laws are commutativity of conjunction and monotonicity of the consequence relation. In addition, we want to rule out programs that leave the realm of logic by producing output, deleting files, opening sockets etc. A disadvantage of this approach is that we are unlikely to come up with an exhaustive list of properties that we would like to see preserved. We can certainly cite very prominent and important ones though. Primary examples of predicates that *break* these properties in general are !/0, assertz/1 and if-then-else. (adopted from [5])

**Reference**

**[1]***A Concise Introduction to Prolog*, Copyright © 1989, 1995, 2012 by David Matuszek

<https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Prolog.html>

**[2]***THE DESCRIPTION LOGIC HANDBOOK: Theory, implementation, and applications*. Franz Baader, Deborah L. McGuinness, Daniele, Nardi, Peter F. Patel-Schneider. Cambridge University Press. 2010

(in-text reference as DL)

**[3]***Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming*. Bruce A. Tate.

(in-text reference as SL)

**[4]** Swi\_Prolog documentation, [http://www.swi-prolog.org](http://www.swi-prolog.org/)

**[5]** Power of Prolog, purity: <https://www.metalevel.at/prolog/purity>