SELF-INTERPRETER FOR PROLOG

BY

ASEEL ALKHELAIWI

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2012

MASTER OF SCIENCE THESIS

OF

ASEEL ALKHELAIWI

APPROVED:

Thesis Committee:

Major Professor Lutz Hamel

Edmund Lamagna

Nancy Eaton

Nasser Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2012

ABSTRACT

The semantics of Prolog programs is usually given in terms of model theoretic semantics. However, this does not adequately characterize the computational behavior of Prolog programs. Prolog implementations typically use a depth-first, left-to-right evaluation strategy based on the textual order of clauses and literals in a program. In this paper we introduce a self-interpreter for Prolog, which is a formalization of the syntax and semantics of Prolog using Prolog. This interpreter is a running program that mimics the depth-first, left-to-right evaluation strategy of Prolog interpreters. This means that, the computational behavior of Prolog is captured by obtaining an operational semantics of Prolog based on the logic + control perspective of Prolog. In addition, this paper explains the important difference between the self-interpretation approach used in this paper and the meta-circular interpretation approach. And how self-interpretation considered a true semantic definition of the object language as it sheds light onto all features of the object language, and does not hide features in the features of the defining language.

ACKNOWLEDGMENTS

This research project would not have been possible without the support of many people. I would like to show my greatest appreciation to Prof. Lutz Hamel. I can't say thank you enough for his tremendous support and help. I feel motivated and encouraged every time I attend his meeting. Without his support, encouragement and guidance this project would not have materialized. I would like to express my love and gratitude to my husband and family for their understanding and endless love, through the duration of my studies.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 JUSTIFICATION OF THE STUDY

Semantics is useful for understanding programs or as a tool for analyzing programs in some programming language. The latter is our main interest. Therefore, in this paper we will define an executable operational semantics for Prolog using Prolog. This approach is called self-interpretation in which Prolog is used to interpret itself. Two questions arise here, the first is why do we look for a new semantics for Prolog even though there exist a number of works on formalizing the semantics for Prolog?(e.g. [4][13][5][2][36]). The answer to this question is that even though all of the previous work covers denotational and operational semantics of Prolog, many of them do not have an executable semantics for Prolog. In this research, we write a Prolog program that reflects the semantics of Prolog, and it's a running program (see appendix A). Now, the second question will arise, which is why do we use Prolog to formalize the semantics? The answer is that because Prolog is rigorously based on first-order logic with a well defined execution model, this makes it suitable for the definition of the syntax and semantics of programming languages. In other words, we use the fact that Prolog has a well defined model-theoretic semantics, the Least Herbrand Model, and also a well defined execution model via the resolution rule. Thus, when defining Prolog with Prolog, we immediately obtain an executable definition of Prolog, in other words, an interpreter for Prolog. Since Prolog has a formal semantics, this interpreter can be viewed as a formal definition of Prolog.

1.2 INTRODUCTION

Prolog is a logic programming language based on first-order logic. Prolog programs are statements in the Horn clause subset of first-order logic. Horn-clauses are sets of literals in the form (A1 and … and An implies B) where A1,.., An and B are literals. Extensions beyond pure Horn clause logic is not considered in this paper. Prolog allows for a declarative style of programming and is essentially logic + control [26]. The logic part is the statement of what the problem is that has to be solved. The control part is the statement of how it is to be solved. Figure 1 indicates the relationship between Prolog and logic programming [23].

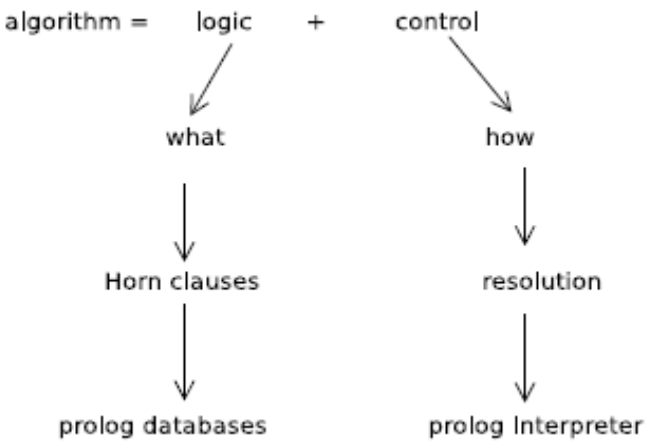

Figure 1: Prolog and Logic Programming Relationship

In this paper, an operational semantics of Prolog is obtained based on the logic+ control perspectives of Prolog, instead of looking at Prolog using the standard first-order logic Herbrand semantic model [37]. In other words, we obtain an interpreter for Prolog written in Prolog which called a self-interpreter for Prolog and it is a working Prolog program.

The language in which the interpreter is written is called the defining language. The other is called the object language [11]. The Prolog self-interpreter behavior needs to match the behavior produced when interpreted by the actual Prolog interpreter. This tells the user about the elegance and the expressiveness of a language.

There is an important distinction between self-interpretation and meta-circular interpretation [34] approaches. The self-interpreter implements the object language features, unlike meta-circular interpreter that uses the existing features, i.e. built-ins, of the object language. As an example, consider unification. Prolog incorporates an operator that implements unification. If we were to map unification in our object language to this operator then we would be doing meta-circular interpretation. However, if we implement unification of the object language in terms of the primitives of Prolog, that is, predicates and terms, then we would obtain a self-interpreter. The difference is very subtle but important: only self-interpreters can be considered true semantic definitions of the object language because they shed light onto all features of the object language and do not hide features in the features of the defining language.

This paper is organized as follow: Chapter Two covers first-order logic and Horn clause logic. Unification, resolution and Herbrand models (and other important aspects) are all explored in this chapter. After that, Chapter Three contains the syntax and semantics of Prolog as well as an example of a Prolog program. Then, Chapter Four contains the self-interpreter and its implementation, the difference between self-interpreters and meta-circular interpreters. Chapter Five presents a number of related works. Finally, we end this paper with Chapter Six which gives some conclusions and further work.

CHAPTER 2: FIRST-ORDER LOGIC AND HORN CLAUSE LOGIC

First-order logic is a considerably richer logic than propositional logic. In propositional logic, atomic propositions are the building blocks for formulas. They are declarative sentences with no internal structure that one can classify as being “true” or “false” such as “Bob is the father of Mary”. Propositions are combined using logical operators that capture notions like “not”, “and”,”or”, etc. In contrast, first-order logic contains elements that allow us to reason about individuals of a given domain of discourse, in addition to the symbols of propositional logic. These elements include function symbols, predicates, and quantification over variables [3].

First order logic has two aspects: One is syntax that is concerned with well-formed formulas admitted by the grammar of the formal language. And the other is semantics that is concerned with the meanings attached to the well-formed formulas and the symbols they contain [27]. Since Prolog programs contain only Horn-clause form sentences, our main interest is the Horn clause subset of first-order logic with no negative knowledge (negative literals). A collection of Horn clauses that do not contain negative knowledge is called definite programs. The main ingredient in definite programs is the inference system, given by the resolution principle. This resolution principle includes the process of making two atomic formulas syntactically equivalent, called unification. The unification process will possibly return new bindings for variables, referred to as substitutions. Substitution is obtaining one formula from another by replacing variables of the original formula by other variables, constants or function symbols.

In this chapter, we introduce the syntax and semantics of first-order logic and Horn clause logic. Then, we explore the important properties in Horn clause logic and more precisely definite programs, which are: resolution, unification and substitution.

2.1 SYNTAX

The syntax of first-order logic consists of logical connectives, quantifiers, and auxiliary symbols, called a fixed part (from (a) to (c) below). And a part that consists of: predicates, functions, variables and constant symbols (from (d) to (g)). [18]
The alphabet of a first-order language consists of the following sets of symbols:

(a) Logical connectives: $\wedge$ (and),
$\vee$ (or),
$\sim$ (not),
$\rightarrow$ (implication),
$\equiv$ (equivalence),
$\perp$ (falsehood);

(b) Quantifiers: $\forall$ (for all),
$\exists$ (there exists).

(c) Auxiliary symbols: "(" and ")".

(d) Variables: A countably infinite set V = {x0, x1, x2, ...}.

(e) Function symbols: A set of symbols with arity>0.

(f) Constants: A set of symbols each of arity zero.

(g) Predicate symbols: A set of symbols with arity$\geq$ 0.

Note that the syntax of first-order logic is in fact a two-sorted ranked alphabet [18]. The sorts are term and formula. The symbols in (d), (e) and (f) are of sort (type) term, and the symbols in (g) and $\perp$ (contradiction) are of sort formula. Formally, we say that every constant and variable is a term, and if t1, ..., tn are terms and f is a function symbol of arity> 0, then f(t1,...,tn) is a term. Furthermore,

- If p is an n-ary predicate symbol and t1,,..., tn are terms, then p(t1,...,tn) is a formula (called an *atomic formula* or, more simply, an *atom).*
- For any two formulas A and B, (A ∧ B), (A ∨ B), (A← B), (A ≡ B) and (∼A) are also formulas.
- For any variable x and any formula A, ∀xA and ∃xA are also formula.

Clauses are formulas in first-order logic in the form:

A1,…,An ← B1,…,Bm (where A1..An and B1…Bn are atoms and m, n ≥ 0)

A1,…,An called the head and B1,…,Bm called the body. Horn clauses are restricted form of first-order logic clauses. They have only one literal in the head, such as:

A ← B1,… ,Bm (where m≥0)

It should be read as "B1 and ..and Bn together imply A". If the body is empty, the clause is called a fact and the implication arrow is omitted. If the head is empty, denoted by the nullary connective "□", then the clause is called the goal clause, or query, and written as

□← B1,… ,Bm (where m≥0)

The literal in the Horn clause could be either a positive literal (atom) or the negative one (the negation of the atom). However, this research is restricted to Horn clauses with only positive literals, called definite clauses, in which a finite set of such clauses is called a definite program.

2.2 SEMANTICS

The meaning of a logic formula is defined as an abstract world called a *structure.* It is either true or false. This means that we need to establish a formal connection between the language and a structure to define the meaning of formulas.

We define the structure, which is the mathematical abstraction of the world, as a nonempty set of individuals, called the *domain*, with a number of relations and functions defined on this domain. [37]

In the language of formulas, constants, function symbols and predicate symbols are the building blocks. So, the link between the language and the structure is established as follows:

An interpretation I of an alphabet A is a nonempty domain D and a mapping that associates:

(a) each constant $c \in A$ with an element $cI \in D$;

(b) each n-ary function symbol $f \in A$ with a function $fI : D^n \rightarrow D$

(c) each n-ary predicate symbol $p \in A$ with a relation $pI \subseteq D \times .... \times D$ *(n times)*

2.3 HERBRAND INTERPRETATION

Definite programs only express positive knowledge, where both facts and rules say when a relation holds, but they do not say when it does not hold. The restriction to definite programs will lead to the elegant model theoretic property where the meaning of programs can be characterized, up to a point, by a single canonical model over ground terms. This model is called the *least Herbrand model.*

Every definite program has a least Herbrand model that reflects all the information expressed by the program and nothing more. It is the intersection of all possible Herbrand models for program P and denoted by Mp. The idea of a Herbrand model is to abstract from the actual meanings of the function symbols of the language, where constants are treated as 0-ary function symbols. A *Herbrand model* for P is a Herbrand

interpretation for P that makes all clauses in P true. To understand what the Herbrand interpretation is, we first need to define the Herbrand universe and the Herbrand base.

The *Herbrand universe* U for a program P is the set of all ground terms formed out of the constants and function symbols appearing in P. The *Herbrand base* B for program P is the set of all ground atoms formed by using predicate symbols from P with ground terms from the Herbrand universe as arguments. An example for the Herbrand universe and the Herbrand base is [37]:

odd(s(0)).
odd(s(s(X))) ← odd(X).

} Prolog program (P)

U(P) = {0, s(0), s(s(0)), s(s(s(0))), …}

B(P) = {odd(0), odd(s(0)), odd(s(s(0))), ….}

Figure 2: Herbrand Universe and Herbrand Base

An interpretation I for program P is a subset of the Herbrand base of P, it is assumed that all atoms in I are true while those not in I are assumed to be false[27]. Formally, Herbrand interpretation is an interpretation I such that:

- The domain of I is U(P) (where P is a definite program).
- For every *c*, c_I is defined to be *c* itself (where $c_I \in$ U(P)).
- For every *n*-ary function symbol *f* the function f_I is defined as follows

  $f_I(x1, ..., xn) = f(x1, ..., xn)$ (where $f_I(x1, ..., xn) \in$ U(P))

  That is, the function f_I applied to *n* ground terms composes them into the ground term with the principal function symbol *f*.
- For every *n*-ary predicate symbol *p* the relation p_I is a subset of $U(P)^n$ (the set of all *n*-tuples of ground terms).

Some Herbrand interpretations for Program P in Figure 2 are:

$I_1 = \emptyset$
$I_2 = \{odd(s(0))\}$
$I_3 = \{odd(s(0)), odd(s(s(0)))\}$
$I_4 = \{odd(s(0)), odd(s(s(s(0)))), \ldots.\}$
$I_5 = B(P)$

Figure 3: Herbrand Interpretation

I_1 is not a model of P as it is not a Herbrand model of odd(s(0)). I_2 is a model for odd(s(0)) but it is not a model of odd(s(s(X))) ← odd(X) since in the instance odd(s(s(s(0)))) ← odd(s(0)), odd(s(0)) ∈ I_2 but odd(s(s(s(0)))) ∉ I_2. Then, I_2 is not a model of P. By using the same instance odd(s(s(s(0)))) ← odd(s(0)), it follows that I_3 is not a model of P. I_4 is a model of P since it is a model for odd(s(0)) and odd(s(s(X))) ← odd(X); let odd(s(s(t))) ← odd(t) be any ground instance of the rule where t ∈ U(P) , if odd(t) ∉ I_4 then odd(s(s(t))) ← odd(t) is true. And if odd(t) ∈ I_4 then it must also hold that odd(s(s(t))) ∈ I_4 .Hence, odd(s(s(t)))← odd(t) is true in I_4. By a similar reasoning, it follows that I_5 is a model of P.

This means that, the Herbrand interpretation is a set of ground facts constructed with the predicate symbols in program P and the ground terms from the corresponding Herbrand domain of function symbols. This is the set of ground atoms supposed to be true by the interpretation [14]. Herbrand interpretation maps every constant to itself. Unlike the interpretation, introduced in the previous section, that maps every constant to some, possibly real world, object.

2.4 SUBSTITUTION

A *substitution* is a finite set of equalities, where "≐" is the equality symbol. The substitution is $\{X1 \doteq t1, X2 \doteq t2, \ldots, Xn \doteq tn\}$ where each Xi is a variable and ti is a term, $X1, \ldots, Xn$ are distinct variables, and Xi is distinct from ti. Each $Xi \doteq ti$ is called a *binding* for Xi. Substitutions denoted typically by the Greek letters θ, γ, σ, possibly subscripted. When some substitution is applied to two terms and it makes them identical, we name this substitution as a *unifier*.

To understand the notion of substitution, consider the following example.

	substitution
`parent(X, Y )` ≐ `parent(Mary, John)`	{X≐ Mary, Y≐ John}
`p(X,Y,Z)` ≐ `p(a, f( W), b)`	{X≐ a, Y≐ f(W), Z≐ b}

Figure 4: Substitution Example

From the example in Figure 4, the substitution makes the two sentences identical. In general, the set $\theta = \{X \doteq a, Y \doteq b\}$ is a substitution, but $\gamma = \{X \doteq a, X \doteq b\}$ and $\sigma = \{X \doteq a, Y \doteq Y\}$ are not since in γ, the same variable X occurs twice on the left hand side of a binding. In σ, the binding $Y \doteq Y$ violates the substitution condition that each Xi must be distinct from ti. [24]

2.5 UNIFICATION

One of the main ingredients in the inference mechanism for definite programs is the process of making two atomic formulas syntactically equivalent. It is called *unification*. It can be expressed as follow: Given two terms containing some variables, find, if it exists, the *simplest substitution* (i.e., an assignment of some term to every

variable) which makes the two terms equal [30]. This substitution is called the *most general unifier*. A unifier is said to be a most general unifier (mgu) of two terms if it is more general than any other unifier of the terms. And we say that substitution θ is *more general* than a substitution σ if there exists a substitution such that $\sigma = \theta\omega$. For example, the two unifiers {X≐ g(Z), Y≐ Z} and {X≐ g(a), Y≐a Z≐a} are considered solutions for the set f(X,Y) ≐ f(g(Z), Z). However, the first unifier is more general than the second one because it didn't specify how Z should be bound.

Finally, we call a set of equations {X1≐t1,, Xn≐tn} a *solved form* if $X1,\ldots,Xn$ are distinct variables none of which appear in *t*1,., *tn*.

For example, consider the following unification problems as posed as queries in Prolog in Figure 5:

```
?- p(X,f(Y)) = p(a,f(b)).
  X = a  Y = b


?- p(X,f(Y),a) = p(a,f(b),Y).
  false


?- p(X,g(Z))= p(m(X),Z).
  X = m(**),
  Z = g(**).
```

Figure 5: Unification Output

In the first case the successful substitution is {X≐a, Y≐b}. But when an attempt is made to unify Y with "a" and "b" in the second query, the result "false" will appear. This is because the instantiated variable cannot be instantiated again-i.e. the variable cannot occur more than once in the unifier. The last query is not in a solved form since

X occurred in m(X) and Z occurred in g(Z). With this query, Prolog will go through a potentially infinite loop. This problem is called the "occur check".

Figure 6 presents the unification algorithm used by Prolog interpreter. This algorithm takes as input a set of equations and returns as output either a solved form equivalent to the set of equations or failure.

Input: A set $\mathcal{E}$ of equations.
Output: An equivalent set of equations in solved form or **failure**.

repeat
  select an arbitrary $s \doteq t \in \mathcal{E}$;
  case $s \doteq t$ **of**
    $f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)$ where $n \geq 0 \Rightarrow$
      replace equation by $s_1 \doteq t_1, \ldots, s_n \doteq t_n$; % *case 1*
    $f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)$ where $f/m \neq g/n \Rightarrow$
      halt with **failure**; % *case 2*
    $X \doteq X \Rightarrow$
      remove the equation; % *case 3*
    $t \doteq X$ where t is not a variable $\Rightarrow$
      replace equation by $X \doteq t$; % *case 4*
    $X \doteq t$ where $X \neq t$ and X has more than one occurrence in $\mathcal{E} \Rightarrow$
      if X is a proper subterm of t **then**
        halt with **failure** % *case 5a*
      else
        replace all other occurrences of X by t; % *case 5b*
  esac
until no action is possible on any equation in $\mathcal{E}$;
halt with $\mathcal{E}$;

Figure 6: Unification Algorithm

We can state the algorithm above informally [36] as, Compare the two terms: (a) If one of the two terms is an uninstantiated variable, i.e. no previous unifications were performed on it, then instantiate it to the other term. Note that, a variable cannot be unified with a term that contains it (*occurs check*, case 5a).

(b) If the two terms are both constants, then if they are the same, succeed otherwise fail.

(c) If the two terms are both function symbols, then check that the function symbols' names and arities are the same and if the parameters can be unified simultaneously, then recursively unify the argument lists.

(d) Otherwise fail.

The algorithm presented in Figure 6 may be very inefficient. One of the reasons is case 5a; the *occur-check*. It gives exponential dependency of the unification time on the length of the structures [37]. Prolog omits the occur-check during unification in order to solve the problem. However, the unification algorithm used in this project is the same as in Figure 6 without omitting case 5a. This is because implementing the unification algorithm without occur-check makes inferencing with resolution unsound. And soundness is a property that is necessary to make sure the conclusion produced by resolution is correct.

To understand the unification algorithm, the built in Prolog operator `'='` will be used to unify two terms as indicated in figure 7.

```
?- a = a.                    % Two identical atoms unify
true.

?- a = b.                    % Atoms don't unify if they aren't identical
false.

?- X = a.                    % Unification instantiates a variable to an atom
X=a

?- X = Y.                    % Unification binds the two variables
X=Y

?- p(a,b) = p(a,b).          % Two identical function symbols unify
```

```
true.

?- p(a,b) = p(X,Y).          % Two function symbols unify if they have
X=a,                         % the same name and the same  arity
Y=b.

?- p(a,Y) = p(X,b).          % Instantiation of variables may occur
Y=b,                         % in either of the terms to be unified
X=a.

?- p(a,b) = p(X,X).          % In this case there is no unification
false.                       % because X will be unified to two different
                             %  values
```

Figure 7: Unification Examples

2.6 RESOLUTION

A deduction method called resolution was developed by John Alan Robinson [38] which was proposed as a uniform proof procedure for proving theorems in Horn clause logic. The algorithm was then refined by Robinson, Kowalski and others [8] [19]. The resolution principle is introduced as a refutation/deduction mechanism for formulas in clausal form. For sets of definite programs, there is a variant of resolution called SLD-resolution. The general idea is to find a refutation for the goal we want to prove. Finding a refutation means that, assuming the goal is false, if the proof of falsity fails then the goal is true.

The fundamental operation of a resolution system takes a pair of clauses as input and produces a new clause as output, called the *resolvent*. In a Prolog computation we have a program which is a set of definite clauses, and a single goal clause- query- which expresses the problem instance we want to solve. In Prolog's resolution method, one of the two clauses we resolve must always be a goal clause, and the resolvent always becomes the new goal clause. This is called *linear resolution*.

The resolution step then proceeds as follows. First, Prolog takes the current goal clause and selects the leftmost member of the goal clause. This is called resolution with a *selection function*. Prolog's selection function always returns the leftmost expression of a goal clause. This sort of resolution system is called *linear resolution with selection function*, or SL-resolution. Therefore Prolog's particular type of resolution called SLD-resolution, with the `D' standing for `Definite Clauses'.

In defining resolution for predicate clauses, we now add an environment, where we keep track of all the substitutions made so far. The *environment* gives us the current value of all of our variables. After selecting an atom from the current goal clause, the next step in the resolution process is to search the program for a clause whose head *unifies* with the selected sub-goal. Then, replace the current sub-goal with the body of that clause. This search follows the order in which clauses appear in the program text. Note that if we resolve the goal with a fact, we simply remove it from the goal and we replace it with nothing as facts are clauses with no bodies. Intuitively, when the goal finally becomes empty, this means that we reduced our original goal to a collection of facts, and so we have proved the original goal, view a refutation. To understand the idea of viewing a refutation, consider the following definite program:

proud(X) ← parent(X, Y), newborn(Y).
parent(X,Y) ← father(X, Y).
father (adam, mary).
newborn(mary).

A refutation of the goal clause (□ ← proud(Z)) is:

a) □←proud(Z)

a.1) Unifies with (proud(X) ← parent(X, Y), newborn(Y).).
a.2) Apply the substitution {X≐Z} to (parent(X, Y), newborn(Y).)
a.3) Replace (proud(Z)) with (parent(Z, Y), newborn(Y)):

b) □←parent(Z, Y), newborn(Y).

b.1) Unifies with (parent(X,Y) ← father(X, Y).).
b.2) Apply the substitution {X≐Z} to (father (X, Y).)
b.3) Replace (parent(Z, Y)) with (father (Z, Y))

c) □←father (Z, Y), newborn(Y).

c.1) Unifies with (father (adam, mary).).
c.2) Remove (father (Z,Y)) since it is unified with a fact.
c.3) Apply the substitution {Z≐adam, Y≐mary}to (newborn(Y)) .

d) □←newborn(mary).

d.1) Unifies with the fact (newborn(mary).) and this leads to a refutation
d.2) Return {Z≐ adam}

e) □ (The empty goal)

In the process of finding a clause whose head unifies with the currently selected atom, we will find the mgu for the clause-head and the goal-atom. The resolvent of the goal with the clause we select is the result of first removing the selected member of the goal clause, then replacing it with the body of the selected program clause, and finally adding the mgu to the environment. Figure 8 below captures the way SLD-resolution generally works.

(a) Given: A goal clause □← G_1,...,G_k and a constraint pool C.

(b) Search the program for a (variant of a) program clause $A \leftarrow B_1,...,B_n$ such that G_1 unifies with A with σ as their most general unifier (mgu). If no such clause exists, then the resolution step fails.

(c) Replace G_1 with $B_1,...,B_n$.

(d) Replace C with the solved form of the system $\sigma \cup C$. Thinking of σ as a substitution function, the new goal equals $\square \leftarrow (B_1,...,B_n,G_2,...,G_k)\sigma$

Figure 8: Prolog's Version of the Resolution Rule

Finally, we say that given a particular computation rule R (selection rule), an SLD-derivation $G \overset{\theta}{\Longrightarrow}^* G'$ (where G is a definite goal) is a sequence of derivation steps $G \overset{\theta 1}{\Longrightarrow} \ldots \overset{\theta n}{\Longrightarrow} G'$ such that $\theta = \theta 1 \ldots \theta n$. An SLD-refutation is obtained where G' is the empty goal □. An SLD- derivation is called failed ($\perp$) if it is not a refutation and the last element of the goal G cannot be resolved with any clause [38][37].

CHAPTER 3: PROLOG

Prolog was invented by Alain Colmerauer and colleagues at Marseille and Edinburgh in the early 70s [44]. It is a declarative language that allows a few control features for acceptable execution performance. Prolog uses the idea of closed world assumption which is negating everything that is not explicitly, or implicitly, in the program. Prolog programs in this paper consist of definite clauses. And to implement a query, Prolog uses the Resolution Principle, introduced before, which is an efficient proof procedure for definite clause logic. In this chapter the syntax and semantics of prolog is discussed as well as the closed-world assumption. Then, we will show a simple Prolog program and show how to prove queries with this program.

3.1 PROLOG SEMANTICS

In Prolog, there are two distinct ways to understand its semantics, procedural and declarative way. The procedural, or operational, semantics describes the way the sequence of states passed through when executing a program. This means that the user can understand a Prolog program as a set of descriptive statements about a problem. This type of semantics is obtained by resolution. In the other hand, the declarative semantics, informally, interprets each term as shorthand for natural language phrases, e.g.:-

[A, B] = "the list whose first element is A and remaining elements are B"

A clause 'P: - Q, R, S.' is interpreted as:-

"P if Q and R and S"

Furthermore, each variable in the clause interpreted should be interpreted too as some arbitrary object. The declarative semantics is obtained by the Herbrand model semantics and specifically by the Herbrand base .This type of semantics recursively defines the set of terms which are asserted to be true according to a program. And one says the term is true if it is the head of some clause instance and each of the goals (if any) of that clause instance is true. And in order to obtain an instance of a clause (or term), Prolog substitutes, for zero or more of its variables, a new term for all occurrences of the variable. Thus the only true instance of the goal:-

reverse ([l,2,3],X) is:-

reverse ([l,2,3],[3,2,l])

So, the declarative semantics gives some understanding of a Prolog program without looking into the details of how it is executed. Unlike the procedural, operational, semantics that describes the way a goal is executed by giving the semantics of Horn-clause programs under resolution with a depth-first search strategy [43]. In this paper, we use both semantics, the declarative semantics via Herbrand base and the operational semantics via resolution.

3.2 WHY PROLOG?

Prolog is a very different programming language when compared to a language such as Java. Prolog is a typeless language such that there are no object-oriented models and selection methods for any denotation as they are embedded in Prolog's unification. In Java, on the other hand, a large collection of classes, interfaces,

methods and a test main method need to be produced before executing anything. In this comparison, the conflict is exposed between flexibility and conciseness on one hand, and security and robustness on the other. In strong contrast to Java and an object-oriented methodology, Prolog appeals to an interactive and incremental type of program development [31]. When using Prolog to write interpreters for programming languages, we rely on the following properties.

- Using rules and unification in Prolog, one can express structurally inductive definitions straightforwardly, e.g.,

  statement(if(A,B),· · ·):- condition(A,· · ·), statement(B,· · ·), · · ·.
- Using Prolog structures, Data types for symbol tables and variable bindings are easily implemented.
- Prolog is an easily accessible framework compared with set and domain theory. Specifications can be developed and tested incrementally and interactively, they also can be monitored in detail using a tracer.
- Prolog has a well defined model-theoretic semantics, the Least Herbrand Model, and also a well defined execution model via the resolution rule. Thus, when defining Prolog with Prolog, an executable formal definition of Prolog (interpreter for Prolog) is immediately obtained.

3.3 SYNTAX

Prolog is a logic programming language based on first-order logic and is essentially logic + control [26]. Prolog can be separated in two parts. First, a Prolog program, sometimes called database, that contains the facts and rules used by the user of the program and contains all the relations that make this program. Second, is the query

where the user can ask questions about relations described in the program. A fact states that a relation holds between individuals unconditionally. It semantically constitutes a declaration of a true state of affairs. A rule states that a relation holds between individuals provided that some other relations hold. It may be used to deduce new facts. A query is a goal for Prolog to try to satisfy. If Prolog is able to find facts and rules that allow it to conclude that the goal is true, we say that the goal is `satisfied' or ‘succeeds’; if a goal cannot be satisfied, we say it `fails'. These facts, rules and queries consist of constants, variables, function symbols and predicates.

- **Constants** are strings of characters starting with a lowercase letter (or enclosed in apostrophes) or strings of digits with or without a decimal point and a minus sign.
- **Variables** are strings of characters beginning with an uppercase letter or an underscore.
- **Structures** consist of a **function symbol**, which looks like an atom, followed by a list of terms inside parentheses, separated by commas. Structures can be interpreted as **predicates** (relations, a truth- valued function such as ‘odd’ and ‘married’) or as **structured objects** (see Figure 9 the structures are depicted as a tree) [39]:

```
likes(john,mary).
male(john).
person(name('Mary','John'),date(December,21,1984)).
```

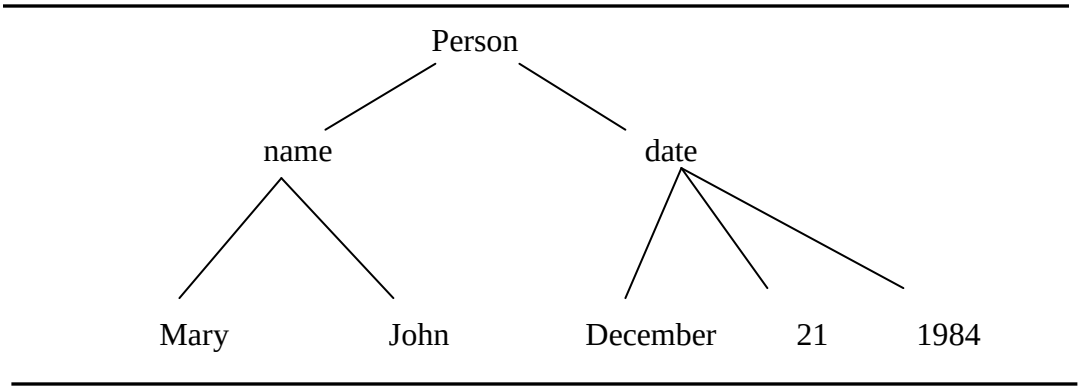

Figure 9: Structure Tree

A Prolog program is a sequence of statements, called Horn clauses, of the form P0 :- P1, P2, …, Pn**.** where each of P0, P1, P2, …, Pn is an atom. A **period** terminates every Prolog clause. A clause can be read as:

P0 is true if P1 and P2 and … and Pn are true. The atoms can be either positive or negative literals. However, we will restrict this research to atoms with positive literal, definite clauses.

In a clause, P0 is called the **head**, and the conjunction of goals P1, P2, …, Pn forms the **body** of the clause. A clause without a body is a **fact**: "P." means "P is true". A clause without a head is a **goal clause** or a **query**, written as "?- P1,P2, …, Pn." or ":- P1,P2, …, Pn." and is interpreted as "Are P1 and P2 and … and Pn true?" or "Satisfy goal P1 and then P2 and then … and then Pn".

To program in Prolog, the user needs to define a database of facts about the given information and rules about how additional information can be deduced from the facts. Then, write a query that sets the Prolog interpreter into action to try to infer a solution using the database of clauses.

3.4 CLOSED-WORLD ASSUMPTION

Prolog constructs a proof to give a positive answer to a query by showing that the set of facts and rules of a program implies that query. Therefore, when Prolog gives

the answer “true” to a query, this means not only that the query is true, but that it is provably true. And when Prolog answers with “false”, this doesn’t mean the query is necessarily false it is just that Prolog failed to derive a proof, it is just not provably true. Negating everything that is not explicitly, or implicitly, in the program is often referred to as the closed world assumption. This is opposite to the *open world assumption*, which means that a term is false only if it can be proven false. Consider the following problem from [35] to see the impact of closed-world assumption. There are three blocks A, B and C arranged as shown:

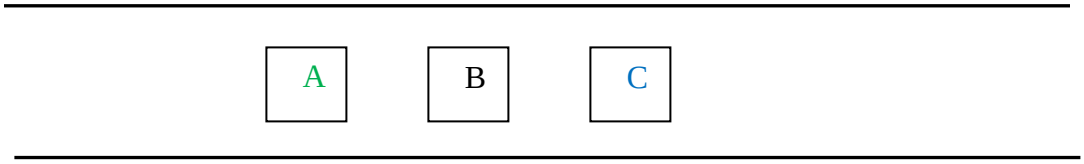

Figure10: Three Colored Blocks

A is a green block, C is blue and the color of B is unknown. In this arrangement of blocks the question is: Is there a green block next to a block that is not green? The answer is yes, because if B is green, it is next to a non-green block C and if B is not green, then it is next to a green block A.

A reasoning system that solves this problem is the one that must be able to ignore whether block B is green or not to infer that some blocks have a certain relation to each other [34]. This is not possible in a system that uses closed-world assumption such as Prolog since it needs to know if B is green or not to infer the relations between the blocks.

Prolog does not distinguish between being unable to find a derivation, and claiming that the query is false; that is, it does not distinguish between “false” and “unknown” values. For example, having the program:

```
male (phil).
female (liz).
parent (phil, chas).
parent (liz, chas).
mother (M,C):- female(M), parent(M,C).
```

Figure 11: Prolog Program

The queries below will be responded by Prolog with “false’:

```
?- male (liz).  %false value

   false

?- female(mary).  %unknown value

   false
```

Figure 12: Prolog Answer

When Prolog responded with “false”, as indicating that a query is false, we are making use of the idea of *negation as failure*: if a statement cannot be derived, then it is false.

Negation as failure is an important feature of Prolog. It is a rule of inference that assumes a fact is false when all possible proofs of the fact being true have failed. Not only does it offer useful expressivity, the ability to describe exceptions, it also offers it in a relatively safe form. In fact, negation as failure comes built in Prolog; we don't have to define it at all. In Standard Prolog the operator “\+”and “not” means negation as failure.

At the first sight, the closed-world assumption seems not applicable and not flexible. But since we restrict this research to definite programs, no negative

knowledge, the outcomes of open-world assumption and closed-world assumption are equivalent. This means that using negative knowledge will make the closed-world assumption not as applicable as open-world assumption.

3.5 PROLOG EXAMPLE AND HOW IT WORKS

Below is an example of a Prolog program:

```
male (phil).
female (liz).
parent (phil, chas).
parent (liz, chas).
mother (M,C):- female(M), parent(M,C).
?-mother (X,chas).
?-mother (liz,chas).
```

Figure 13: Prolog Program and Queries

The first four lines of the program are facts. The first fact can be interpreted as "Phil is a male," and the third fact can be interpreted as “Phil is a parent of Chas.” The fifth line is a rule that defines the relation: *"mother of"* and can be interpreted as “M is a mother of C if M is a female and M is a parent of C.” The last two lines in the program are queries. The goal implied by the queries is to identify, if possible, the mother of Chas and if Liz is a mother of Chas. So, the set of facts and rules is referred to as the *database* for the described problem. When a query is entered into the system, Prolog's underlying deductive mechanism- resolution- infers an answer using the given inference rules on the facts and relations. Which means that in the first query of the program above, Prolog will return Liz and in the next one it will return “true”.

Prolog attempts to prove the goal (`?-mother (liz,chas).`) by first proving `female(liz)` and then (if this is successful) by proving `parent(liz, chas)`. If

this process fails at any point, Prolog will report its failure with “false”. This means that the deduction algorithm used by Prolog proceeds in a left-to-right, depth-first order. To show that, this is how Prolog evaluates the goal (`?-mother (liz,chas).`):

```
?- mother (liz,chas).
 1 1 Call: mother (liz,chas) ?
 2 2 Call: female (liz) ?
 2 2 Exit: female (liz) ?          %success
 3 2 Call: parent (liz,chas) ?
 3 2 Exit: parent (liz,chas) ?     %success
 1 1 Exit: mother (liz,chas) ?     %success
 true
```

Figure 14: Trace

In the other goal (`?-mother(X,chas).`) Prolog will attempt to satisfy the query by searching sequentially for the rule with head `mother.` The goal (`?-mother(X,chas).`) and the head of the rule (`mother (M,C):- female(M), parent(M,C).`) have the same name and can be made syntactically identical, or unified, by the substitution or unifier { M≐X , C≐chas}. If this substitution is made in the body of the rule, the result is:

```
?- female(X), parent(X,chas).
```

What is needed is a female, X, and then a parent X of chas. (`female(X)`) can be unified with the fact (`female(liz).`) by using the unifier {X≐liz}. Then, this substitution will be used too for (`parent(X,chas).`) since it has the same variable as `female` ,X.

CHAPTER 4: SELF-INTERPRETER

Our interpreter for Prolog is implemented as a Prolog program. This means that, we will evaluate Prolog programs using an evaluator -interpreter- that is itself implemented in Prolog. Since program execution is a series of steps to prove some goal, we can describe program execution using Prolog, which is our tool for theorem proving [1]. The implementation of the interpreter depends upon procedures that define the syntax of the atoms to be evaluated. A parser is used to convert the program entered by the user to a parsed program with a representation acceptable by the interpreter. The Prolog interpreter has three distinct phases:

(a) Scanner: Scans a Prolog file that contains the user program and the query that needs to be proven, and produces the appropriate tokens, the building blocks that describe the program.

(b) Parser: Converts the tokens to a parse tree. This parse tree contains two lists, the first is the list of the facts and rules in the user program written in a special form, called the database. The other is the query.

(c) Interpreter: Interprets the query using the database and produces the appropriate output.

It is important to note that this interpreter is a self-interpreter which is different than a meta-circular interpreter used in [40][1]. In this chapter, the self-interpretation approach and meta-circular approach are defined as well as the difference between them. Then, the implementation of the self-interpreter (scanner, parser and interpreter) is proposed.

4.1 SELF-INTERPRETER FOR PROLOG

An interpreter (or evaluator) for a programming language is a procedure that performs the actions required to evaluate an expression of the programming language [1]. The interesting thing about an interpreter, which determines the meaning of expressions in a programming language, is that it is just another program that can be interpreted by another interpreter. One advantage of expressing the semantics as a program is that we can run it to gives us a working model of how Prolog itself evaluates queries. So, as Harold and Gerald in [1] say is that in writing the interpreter they consider themselves as designers of languages, rather than only users of languages designed by others. Recall that, the language in which the interpreter is written called the defining language. The other is called the object language [11]. In our case, where the interpreter language and the language the interpreter evaluates are both Prolog, the object language and the defining language coincide.

In this paper we define the syntax and semantics of Prolog using Prolog. The interesting ramification of this approach is that instead of looking at Prolog using the standard first-order logic Herbrand semantic model [37], we will obtain an operational semantics of Prolog based on the logic + control perspective of Prolog. In other words, we obtain an interpreter for Prolog written in Prolog. This approach to programming language definition is typically called "self-interpretation." [34].

Self-interpretation needs to meet two requirements: First, the behavior produced by the self-interpreter when interpreting programs must be the same as the behavior produced when interpreted by any other interpreter, i.e. the two interpreters must produce the same output for any legitimate input, though not necessarily at the same

speed. Second, the self-interpreter must not use language features [31] because this is considered meta-circular interpretation not self-interpretation.

It is worth mentioning that our approach to self-interpretation is different from the approach commonly referred to as "meta-circular interpretation" [34] in that we will use Prolog to actually implement the features of Prolog rather than just mapping them to existing features in Prolog. Self-interpreters, unlike meta-circular interpreters, can be considered true semantic definitions of the object language because they do not hide features in the features of the defining language.

Prolog is particularly well suited for defining semantics. This is because it is rigorously based on first-order logic with a well defined execution model that makes it suitable for the definition of the syntax and semantics of programming languages. Observe that any semantic definition of some programming language using Prolog immediately becomes a formal specification of the behavior of that language, because Prolog itself is rigorously based of first-order logic. This formal definition of the syntax and semantics of a programming language can then be used for the validation and verification of programs written in the object language. In this case, our perspective is shifted slightly and instead of viewing Prolog as a programming language we view it as a theorem prover [42]. Note that, since Prolog is based on Horn-clause sets, we will restrict our semantics to evaluate Horn clause logic, specifically definite programs (no extra-logical features, such as cut "!", will be covered in this research).

4.2 META-CIRCULAR INTERPRETER FOR PROLOG

A meta-circular interpreter is a special case of a self-interpreter. It uses the same programming language for both the object language and the defining-language. The difference is that instead of implementing the object-language, say Prolog, features, it applies the existing features of the object-language's interpreter to the code being interpreted, in contrast to self-interpreters. The simplest Prolog meta-circular interpreter is the following program:

```
solve(Goal):-call(Goal).
```

Figure 15: Simple Meta-circular Interpreter

The interpreter in Figure 15 just calls the Prolog built-in interpreter without taking any action. As a result, there is no advantage of using such an interpreter as a defining tool. The predicate rule `solve` takes as input the query, which the user want to prove, and then calls the built-in predicate rule `call` that lets the Prolog interpreter proves the goal. Another meta-circular interpreter is "vanilla meta-interpreter" [22], see Figure 16. This interpreter uses Prolog's built-in unification by the use of the predicate `clause(A, B)` which unifies the goal A with a head of some clause in the program and returns its body B. However, this interpreter gives access to Prolog's database search engine for example; the user can change the order in which the goals are executed.

```
solve(true).

solve((A,B)):-
    solve(A),solve(B).
```

```
solve(A):-
    clause(A,B),solve(B).
```

Figure 16: Vanilla Meta-circular Interpreter

The first `solve` predicate in Figure 16 checks to see if its argument is a "true" fact. The second `solve` checks if the argument of the solve predicate is the conjunction of two predicates. This `solve` can handle any number of goals recursively by calling `solve` to the first goal and then calling it to the set of remaining goals. Finally, the third `solve` will be reached when the two previous attempts have failed. It will call the built-in meta-predicate `clause` to return the body of the rule in which its head unifies with the goal, then it calls `solve` to evaluate the body of that rule (if there is no body, then B=true) [29]. The `solve` predicate implements the same left-to-right, depth-first search as the built-in prolog interpreter. However, it didn't implement any built-in features of Prolog, i.e. unification, it just map them to those features.

Meta-circular interpretation is easy to explain and understand. It is a ready-to-use implementation that is accessible. These kinds of interpreters can be useful starting points for the development of debuggers, tracers, profilers, and they are used for extending languages or for changing aspects of semantics [33].

There are a number of papers that used meta-circular interpretation. One of these papers is [33] where the authors based their work on meta-circular interpretation. They illustrate a generic approach to the abstract interpretation of Prolog. The method they present was a continuation of work done by Codish and Demoen in [10]. They consider an interpreter which corresponds to bottom-up semantics, instead of top-down semantics. Such an interpreter generates new facts incrementally, in the style of the well-known *TP* operator [17] or the *s*-semantics [15].

The fundamental operation in the bottom-up semantics of a program *P* is the repeated application of facts derived so far, generating new instances of their heads to solve the bodies of *P*'s rules. Codish and Demoen [10] captured this idea with the simple meta-circular interpreter shown below.

(a) The control

```
iterate :- operator, fail.
iterate :- retract(flag), iterate.
iterate.
record(F) :- cond_assert(F).
raise_flag :- ( flag -> true ; assert(flag) ).
%% if flag return true else assert(flag)
cond_assert(F) :-
\+ (numbervars(F,0,_), fact(F)), assert(fact(F)), raise_flag.
%%" \+" returns true if (numbervars(F,0,_), fact(F)), assert(fact(F)) cannot be proven
```

(b) The logic

```
operator :- my_clause(H,B), prove(B), record(H).
prove([B|Bs]) :- fact(B), prove(Bs).
prove([unify(A,B)|Bs]) :- unify(A,B), prove(Bs). prove([]).
```

(c) Using the interpreter

```
go(File) :-
load_file(File),
iterate,
showfacts.
```

```
showfacts :-
fact(F),
numbervars(F,0,_),
print(F), nl,
fail ; true.
```

The interpreter above is divided conceptually into three components [33]. **(a)** The **control** component which triggers iteration of an operator until no new facts are derived. **(b)** The **logic** component contains the predicate "`operator`" which provides the inner loop of the algorithm. This predicate proves the body of a clause and adds the head of the same clause to the set of facts derived so far, provided it is a new fact. **(c)** The **interpreter** component which contains the predicate "`go`" that facilitates the use of the interpreter. This predicate loads the program to be interpreted, initiates iteration and finally prints the derived facts on the screen. A serious drawback of the

bottom-up evaluation is that it focuses only on the provable ground atoms. However, the authors consider the meta-circular approach they used is a strong approach since it can produce fast analyzers in very short development time.

John McCarthy in [32] introduces the concept of meta-circular interpreter and gives the first implementation for such an interpreter in the LISP programming language. He represents the features of the LISP interpreter by defining a universal function written in LISP. This function takes as input the definition of the LISP function together with the arguments list. Then, it uses these arguments to evaluate the function. Another important meta-circular interpreter for Lisp is 3-Lisp in [40], which is an elaboration of McCarthy's original proposal.

4.3 IMPLEMENTATION

4. 3. 1 SCANNER

The scanner (sometimes called tokenizer) reads the file containing the program text and produces a list of tokens according to the syntax of Prolog. In other words, tokenization is the process of breaking a text file up into words and/or other significant units such as special characters [12]. For example, the tokenizer will break the input program below:

```
father(adam,chris).

grandfather(X,Y):- father(X,Z),father(Z,Y).
```

Into a list of tokens:

```
[father, '(', adam, (','), chris, ')', '.', grandfather, '(',
'X', (','), 'Y', ')', :, -, father, '(', 'X', (','), 'Z', ')',
(','), father, '(', 'Z', (','), 'Y', ')', '.']
```

Words and numbers are represented as lists of atoms, special characters stand by themselves. The white space (blanks, line breaks etc.) between the tokens and apostrophe" ' " are skipped.

The scanner will take as input a Prolog file (with .pl extension). This Prolog file contains a Prolog program, which consists of facts and rules, and ends with a query. For example, a file named "input.pl" which contains the following:

```
father(adam,chris).
father(chris,bob).                                  } % The program
grandfather(X,Y):- father(X,Z),father(Z,Y).
?-father(adam,A).                    % A query about the program
```

Figure 17: Prolog Program and Query

The scanner reads the file and converts the program and query above to a list of character codes, ASCII codes. After that it will start tokenizing these codes to:

- Word or numbers
- Letter
- Colon (**:**).
- Period-dot (**.**)
- Hyphen (**-**)
- Left-parentheses (**(**)
- Right-parentheses (**)**)
- Comma (**,**)
- Left-bracket (**[**)
- Right-bracket (**]**)
- Question mark (**?**)

- List operator (|)
- Semi-colon (;)

So, the result of scanning the example in Figure 17 is:

```
[father, '(', adam, (','), chris, ')', '.', father, '(', chris,
(','), bob, ')', '.', grandfather, '(', 'X', (','), 'Y', ')',
:, -, father, '(', 'X', (','), 'Z', ')', (','), father, '(',
'Z', (','), 'Y', ')', '.', ?, -, father, '(', adam, (','), ‘A’,
')', '.']
```

Figure 18: List of Tokens

This list of tokens will be fed to the parser. Note that the query is in the same file as the Prolog program. This is because after parsing and producing the parse tree the interpreter will try to prove this query in the context of that program.

4. 3. 2 PARSER

After scanning the Prolog file, the output of the scanner which is a list of tokens is the input to the parser. The parser is an LL(1) parser [28]. This parser is top-down or a recursive descent parser that requires a one token look-ahead. LL(1) means that the parser is processing the input string from *left to right* (first L) using *leftmost derivations* (second L). Scanning the input file from left to right is exactly what we did in the previous section. In other words, these parsers scan the input file from left to right and try to match them against the terminals of the grammar [21]. However, what we did is slightly different. Instead of sending the token to the parser as soon as we scan one, we just scan the whole file from left to right and then send the list of tokens

to the parser. Then, the parser starts matching them against the terminals of the grammar.

Since the grammar rules in LL(1) parsers get translated to recursive function calls, this makes them easier to write, understand and debug. These reasons helped in choosing LL(1) parsing for this project.

The syntax of Prolog is captured by the following Context-Free Grammar (CFG) that we developed:

PROGRAM → RULELIST QUERY

RULELIST → RULE . X

X → RULELIST | ε

RULE→ ATOM Y

Y → :- ARGUMENTLIST | ε

ARGUMENTLIST→ATOM Z

Z→ , ARGUMENTLIST | ε

ATOM → PRED (PREDLIST)

PREDLIST → T M

M→ , PREDLIST | ε

T → id N | [LL]

N→(PREDLIST) | ε

PRED→id

LL→TT MM | ε

MM→, LL | | LLL | ε

TT→[K | id

K→] | LL]

LLL→ id | LLL | [ ]

QUERY → **:-** BODYLIST | **?-** BODYLIST

Figure 19: LL(1) Grammar for Prolog

This grammar is LL(1) grammar with no left-factoring and no left recursion. A grammar with left-factoring is in the form: (A→ a B, A→ a C) where A, B and C are non-terminals and "a" is a terminal. Left recursion is in the form (A→ A β) where A is a non-terminal and β can be either a terminal or a non-terminal.

The grammar in Figure 19 is written as a LL(1) parser. This parser produces the database and query, written in a special form, for the interpreter. Recall that our parser will receive as input the tokens list. Consider the program in Figure 17, we will give the parser the token list in Figure 18, and the output is:

```
% DATABASE
[predicate(father, [adam, chris]), predicate(father, [chris,
bob]), predicateRule(grandfather, ['X', 'Y']), body(father,
['X', 'Z']), body(father, ['Z', 'Y'])],
% QUERY
[query(father, [adam, 'A'])]
```

Figure 20: Parser Output

The output is two lists, the first one considered the database in our interpreter and the other is the query that the interpreter will try to prove. Note that, the fact is written as `predicate` and the predicate rule written as `predicateRule`.

The first three rules in Figure 19 are written in Prolog as:

```
program ([Id|Tokens],Parselist,[],Oparsetree,Query):-
      ruleslist([Id|Tokens],Listsofar,[],Parsetree),
      query(Listsofar,Parselist,[],RQuery).   %predicte set={id}

ruleslist([Id|Tokens],L,Iparsetree,Oparsetree):-
      rule([Id|Tokens],L1,Iparsetree,Parsetree),
      matchperiod(L1,L2),
      x(L2,L,Parsetree,Oparsetree).           %predicte set={id}

matchperiod(['.'|Tokens],Tokens).

x(['-'|Tokens],['-'|Tokens],Parsetree,Parsetree).
x(['?'|Tokens],['?'|L],Parsetree,Oparsetree):-
      x(Tokens,L,Parsetree,Oparsetree).   %predicte set={id,?-,:-}
x([':'|Tokens],[':'|L],Parsetree,Oparsetree):-
      x(Tokens,L,Parsetree,Oparsetree).
x([Id|Tokens],L,Iparsetree,Oparsetree):-
      rulelist([Id|Tokens],L,Iparsetree,Oparsetree).
```

Figure 21: A Snippet of the Parser Program

Note that, every rule will expect to see specific tokens in order to apply the rule. These specific tokens are called predicate sets. This means that, every rule or non-terminal (such as PROGRAM, RULELIST etc) has its own predicate set which every non-terminal will expect to see in the token list. These predicate sets are covered in appendix C. However, the predicate sets for the subset of the parser in Figure 21 appears in front of every rule as a comment.

4. 3. 3 INTERPRETER

As the database and the query produced by the parser, the interpreter tries to prove the query using the facts and rules in the database. Figure 22 is an algorithm called `semantics`, which takes a goal and attempts to prove it [20]. If a solution is found, then variable bindings are printed and the machine stops its search. But if there are no

variables in the goal, it simply prints “true” and halts the search. If no solutions are found, “false” is printed.

```
Algorithm: semantics(S, V)

S: The query to solve is first in this stack, then after that
the sub-goal pushes into the stack to prove

V: A list of variables binding to print upon finding a
solution

if(empty(S))
print_solutions(V)

goal = pop(S)
predicate = lookup_predicate_head(goal)

while( predicate != NULL)
if(unify(head(predicate), goal))
     push(S, body(predicate))
     break
         else
          predicate = lookup_predicate_head(goal)


semantics(S, V)
     return true
```

Figure 22: Proving Goal Algorithm

The algorithm is fairly simple. Essentially, we pop a goal off of the stack and then look it up in the program (database). Note that, when the program starts, there is only one query that is entered by the user and then, when going through the algorithm, the sub-goals will be placed in this stack. If the first predicate that we want to unify with does not match (possibly, due to different arities), the algorithm will search for another predicate that matches the head of the goal (this process is called backtracking), but if the predicate matches our goal then the head of the predicate is unified with the goal and replaces the goal on the stack with the (possibly empty) list of sub-goals provided by the predicate. Next, we do the same thing recursively. If the stack is empty, the

solution is found and then printed for the user. The way this algorithm written in the interpreter is captured in Figure 23. Note that if the argument proceeds with "+", then it is an input and if it proceeds with "-"then it is an output:

```
sem(+<query>,+<database>,+<inputlistforbacktracking>,
 -<outputlistforbacktracking>,+<inputsubstitutions>,
-<answersubstitutions>):-

searchDB(+<query>,+<database>,-<matchingpredicate>),

insertPredforBT(+<matchingpredicate>,+<inputlistforbacktracking
>,-<backtrachinglist>),

unify( +<query>,+<matchingpredicate>,-<unifiedpredicate>,
+<inputsubstitutions>,-<substitutions>),

checkforvar(+<unifiedpredicate>,-<flag>),

checkflag(+<flag>,+<unifiedpredicate>, +<database>,
+<backtrachinglist> ,-<outputlistforbacktracking> ,
+<substitutions> , -<answersubstitutions>),

displayResult(+<answersubstitutions>).
```

Figure 23: `sem` Predicate Rule in the Interpreter

The first argument in the `sem` predicate in Figure 23 is the query (`<query>`) written in the form (`[query(X,List)]`), where `X` is the name of the query and `List` is the argument(s) list of that query. The second argument, `<database>`, contains the database (such as Figure 20, the database part). `<inputlistforbacktracking>` is the input backtracking list that tells the interpreter where it stops during the evaluation in order to backtrack, if needed. It is initially an empty list ([ ]). `<outputlistforbacktraching>` is the output backtracking list. `<inputsubstitutions>` is the input binding list; initially it is an empty list. `<answersubstitutions>` is the output binding list and, in this

`sem` predicate particularly, it is the result variable bindings (if any) of the program we want to interpret.

Example: Using the parser output in Figure20, the head of the `sem` predicate will look like:

```
sem([query(father, [adam, 'A'])], [predicate(father, [adam,
chris]), predicate(father, [chris, bob]),
predicateRule(grandfather, ['X', 'Y']), body(father, ['X',
'Z']), body(father, ['Z', 'Y'])], [], BackTracklist,[],Oterms).
```

Then `searchDB` predicate is called, which is a left to right depth-first search that returns the atom in which its head unifies with the query:

```
searchDB([query(father, [adam, 'A'])], [predicate(father,
[adam, chris]), predicate(father, [chris, bob]),
predicateRule(grandfather, ['X', 'Y']), body(father, ['X',
'Z']), body(father, ['Z', 'Y'])],Predicate).
```

`Predicate` is unified with `[predicate(father, [adam, chris])]`, since it has the same name as the query, the same arity and the first argument is `adam`. After that, `[predicate(father, [adam, chris])]` is entered into the backtracking list in case we need to backtrack, using `insertPredforBT` predicate. Then, `unify` predicate is called:

```
unify([query(father, [adam, 'A'])], [predicate(father, [adam,
chris])],UnifiedPred,[],Uterms).
```

The unify predicate will unify the query(`query(father, [adam, 'A']))` with the fact (`predicate(father, [adam, chris])`) and then returns `UnifiedPred= [predicate(father, [adam, chris])]` and `Uterms = [[o,'A',chris]]`

Finally, `UnifiedPred` is checked for variables. If it contains any variable, `sem` predicate is called again but the new `sem` predicate is different than the `sem` predicate

used earlier. If `UnifiedPred` contains no variables, all variables are unified (have bindings), then the result is displayed by `displayResult` predicate which is `A=chris.`

4.3.3.1 UNIFICATION

Executing the unification algorithm is one of the necessary components, besides resolution, to build a Prolog self-interpreter. Below is the `unify` predicate that unifies two argument lists, where the two argument lists do not contain lists such as [‘A’, [1,2], mary]:

```
unify([query(Name,ArgumentList)],[predicate(Name,ArgumentList1)
],[predicate(Name,UnifiedArgumentList)],IUterms,OUterms):-
ulist(ArgumentList, ArgumentList1, UnifiedArgumentList,
IUterms, OUterms).
```

The `unify` predicate always calls `ulist` predicate to start unifying the lists of arguments. All of the `ulist` predicates will start by checking if the two terms, which it attempts to unify, are variables or constants. Using the same example in Figure 20, the `unify` predicate will look like:

```
unify([query(father, [adam, 'A'])], [predicate(father, [adam,
chris])],[predicate(X,Ulist)],[],OUterms).
```

Then `ulist` predicate is called:

```
ulist([adam, 'A'], [adam, chris],[adam|Ul],[],OUterms).
```

`ulist` will try to unify “adam” with “adam”, but since both are constants it will skip this step. Then `ulist` will look like:

```
ulist(['A'], [chris],Ul,[],OUterms).
```

This `ulist` predicate will check if 'A' was unified before (e.g. has a substitution). If yes, this step is skipped and 'A' will not be unified with `chris` and, if not , `ulist`

will unify `'A'` with `chris` and returns `Ul=[chris]` and `OUterms=[[o,'A',chris]]`.

The substitution with “o” written before the substitution is to indicate that a variable is unified with a constant. If the `ulist` predicate is attempting to unify two variables, the substitution is preceded by “v”. And, if `ulist` predicate is trying to unify the variables in the body of a predicate rule with substitutions from the head, the substitution is preceded by “i”.

In the case where the two argument lists in `ulist` contain lists, the unifying is similar to the idea of unifying argument lists that contain no lists. The difference is that it will do more recursion in order to unify the whole list. Furthermore, when the `ulist` predicate will skip a unification of a variable with a term that contains the same variable as argument (the occur check).

4.4 TESTING

The Prolog self-interpreter shows successful results when tested with a number of programs. One of these programs is the interpreter itself. We fed the interpreter to the interpreter and it shows the right results. Therefore, we believe that all the features in our interpreter are true since it gives the right results when the interpreter interprets itself. Figure 24 shows the idea of self-interpretation.

Assume: SCAN is the scanner, PARSER is the parser and INTERP is the Horn clause logic interpreter. Then, for some program P, in pure Horn clause logic, the full interpretation is:

P →SCAN →PARSER → P' → INTERP → (results)

(Where P' is the abstract representation, the parsed program, of the input program P). To prove that the interpreter is a self-interpreter, we did this:

(P' → INTEPR) → SCAN → PARSER → INTEPR → (results)

Figure 24: Self-interpretation

CHAPTER 5: RELATED WORK

A number of approaches to the definition of the semantics of Prolog exist. The most traditional is the fixpoint semantics developed by Herbrand [37], usually referred to as the "Least Herbrand Model."In [4] the authors proposed a logical semantics for pure Prolog (without extra-logical features and negation) based on a four-valued logic. Their semantics enjoys the nice properties of the declarative semantics of logic programming (existence of the least Herbrand model, equivalence of the model-theoretic and operational semantics). It is worth mentioning that their semantics is truly logical for propositional Prolog, whereas it loses part of its logical flavor when moving to the non-propositional case. This is due to the evaluation of existentially quantified goals. This evaluation is based on a suitable ordering on ground instances of goals, which is obtained by exploiting the fixpoint approach of [5]. The problem with the semantics developed here is that it doesn't cover the normal Prolog program (i.e. Prolog programs with negation as failure). So, it needs to be extended to cover the normal Prolog Program as well as the extra-logical features of Prolog.

In [39] the authors develop a denotational semantics that captures the computational behavior of Prolog .They believed that the semantics of Prolog programs need not to be given in terms of the model theory of first order logic because this does not adequately characterize the computational behavior of Prolog programs. So, they developed their semantics based on the fact that Prolog implementations typically use a sequential evaluation strategy based on the textual order of clauses and literals in a program, as well as non-logical features like ''cut''. The authors believe that while the model theoretic semantics is very useful for understanding Prolog programs, it is

necessary to resort to a denotational description in order to reason about tools that manipulate and transform Prolog programs.

The semantics for Prolog in [13] is different than the semantics in [25] in two points: First, in [13] the semantic definitions are motivated by the need to justify program analysis and transformation methods, unlike [25] where their definitions are driven by the goal of generating correct Prolog interpreters. Second, in [13] they give a continuation semantics that models ‘‘cut’’ in a more intuitively accessible manner, but in [25] ‘‘cut’’ is a modeled by means of a special token.

The authors in [41] propose operational and denotatioal semantics for Prolog. They capture the control rule of Prolog and the cut operator. Their donotational semantics provides a goal-independent semantics. Which means that the behavior of a goal in a program is defined as the evaluation of the goal in the semantics of the program. Their approach deals with negation as failure that can be implemented through a clever use of cut operator. They believe their denotation semantics can be used as an effective base for precise program analysis. And the denotational semantics is better than the operational semantics if one is concerned with goal-independent global analysis. Unlike the previous approach in [41], the paper [7] presents a denotational semantics developed for abstract interpretation but it follows a goal-dependent approach.

The denotational Semantics of Prolog is expressed in Algol-68 in [2]. The result is a formal definition that is also executable. Like the parser approach presented in this paper, the tree of the semantics was built in a recursive-descent parser. The authors believe that their interpreter is a reference implementation and is very useful for experimentation.

CHAPTER 6: CONCLUSION AND FURTHER WORK

In this paper we have established the executable operational semantics for Prolog as a self-interpreter for Prolog. The interpreter goes through three stages: scanning, parsing and interpreting. We saw that proving some goal in a Prolog program is nothing but implementing the SLD-resolution on Horn-clause sets (definite programs). The objective of defining a Prolog self-interpreter has been achieved and the interpreter shows successful results when tested with a number of programs. One of the programs used to test the interpreter was the interpreter itself. We fed our interpreter to the interpreter and it showed the right result.

The interpreter is subject to future extensions. These extensions contain:

(a) Cut operator (**!**) that used to prevent unwanted backtracking.

(b) Semicolon (**;**) which basically tells the interpreter to return all the solutions, not just the first one.

(c) Remove the extra bindings appeared in the result when interpreting some programs.

APPENDICES

APPENDIX A: INTERPRETER

```
sem([query(X,List)],DB,BTList,OBTList,IUterms,OUterms):-
returnvar(List,Var),
searchDB([query(X,List)],DB,Predicate),
insertPredforBT(Predicate,BTList,IBTList),
unify([query(X,List)],Predicate,UniPred,IUterms,Uterms),
checkforvar(UniPred,Flag),
checkflag(Flag,UniPred,DB,IBTList,OBTList,Uterms,OUterms,Var,_)
, displayResult1(OUterms,OUterms,Var).


%%%%% semantics of facts %%%%%
sem([predicate(X,List)],DB,BTList,OBTList,IUterms,OUterms,Var,V
ar1):-
unifypredicate(List,Ulist,IUterms,IUterms,Uterms),
checktoplevelvar(Uterms,Var,Flag1),
checktopflag(Flag1,[predicate(X,Ulist)],List,DB,BTList,
OBTList,Uterms,OUterms,Var,Var1).

%%%%% semantics of rule %%%%%
sem([predicateRule(X,List)],DB,BTList,OBTList,IUterms,OUterms,V
ar,Var1):-
searchforbody([predicateRule(X,List)],DB,Body),
sembody(Body,DB,BTList,OBTList,IUterms,Uterms,Var,Var1),
unifywithhead(Uterms,Var,Var1,OUterms).

%%%%% semantics of the body of the rule %%%%%
sem([body(X,List)],DB,BTList,OBTList,IUterms,OUterms,Var,Var1):
-
searchDB([query(X,List)],DB,Predicate),
checkfail(Predicate,[query(X,List)],DB,BTList,OBTList,IUterms,O
Uterms,Var,Var1).

checkfail(fail,_,DB,[[body(X,List)]|Rest],OBTList,IUterms,OUter
ms,Var,Var1):-
sem([body(X,List)],DB,Rest,OBTList,IUterms,OUterms,Var,Var1).

checkfail(fail,_,DB,[[predicateRule(X,List)]|Rest],OBTList,IUte
rms,OUterms,Var,Var1):-
sem([predicateRule(X,List)],DB,Rest,OBTList,IUterms,OUterms,Var
,Var1).

checkfail(Predicate,[query(X,List)],DB,BTList,OBTList,IUterms,O
Uterms,Var,Var1):-
insertPredforBT(Predicate,BTList,IBTList),
unifybody([body(X,List)],[body(X,Ulist)],IUterms),
```

```
unify([query(X,Ulist)],Predicate,UniPred,IUterms,Uterms),
checkforvar(UniPred,Flag),
checkflag(Flag,UniPred,DB,IBTList,OBTList,Uterms,OUterms,Var,Va
r1).
sem([body(string_to_list,[List1,List2])],_,BTList,OBTList,IUter
ms,OUterms,_,_):-
insertPredforBT([body(string_to_list,[List1,List2])],BTList,OBT
List),
call(string_to_list,List1,List),
insert(List,List2,IUterms,OUterms).

sem([body(write,[List])],_,BTList,OBTList,IUterms,IUterms,_,_):
- unifybody1([body(write,[List])],[body(write,[UList])],
IUterms),
insertPredforBT([body(write,[UList])],BTList,OBTList),
call(write,UList).

checktopflag(false,_,_,_,_,_,Uterm,Uterm,_,_).

checktopflag(true,[predicate(X,Ulist)],List,DB,BTList,OBTList,U
terms,OUterms,Var,Var1):-
checkforvar([predicate(X,Ulist)],Flag),
checkflag(Flag,[predicate(X,Ulist)],DB,BTList,OBTList,Uterms,OU
terms,Var,Var1).

unifywithhead(Terms,Var,_,[[o,Var,Result2]|Terms]):-
searchwhengoback(Var,Terms,Result1,true),
search(Result1,Terms,Result2).

unifywithhead(Terms,_,Var1,[[o,Var1,Result2]|Terms]):-
searchwhengoback(Var1,Terms,Result1,true),
search(Result1,Terms,Result2).

returnvar([],false).

returnvar([V1|Rest1],Var):-
valterm1(V1,F),checkF(F,V1,Rest1,Var).

checkF(true,V,_,V).
checkF(false,_,Q1,Var):-returnvar(Q1,Var).

checktoplevelvar([],_,false).
checktoplevelvar([[o,V,_]|_],V,true).
checktoplevelvar([_|R],V,U):-checktoplevelvar(R,V,U).

insert(List,List2 ,L,[[o,List2,List]|L]).

checkflag(true,UniPred,DB,IBTList,OBTList,Uterms,OUterms,Var,Va
r1):- sem(UniPred,DB,IBTList,OBTList,Uterms,OUterms,Var,Var1).

checkflag(false,_,_,_,_,Uterms,OUterms,_,_):-
finalterm(Uterms,OUterms).
```

```
sembody([],_,BTList,BTList,Uterms,Uterms,_,_).

sembody([Body|Rest],DB,BTList,OBTList,Terms,OUterms,Var,Var1):-
insertPredforBT([Body],BTList,IBTList),
unifybody([Body],UniBody,Terms),
sem(UniBody,DB,IBTList,IIBTList,Terms,Uterms,Var,Var1),
sembody(Rest,DB,IIBTList,OBTList,Uterms,OUterms,Var,Var1).

searchforbody([predicateRule(X,List)],[body(_,_)|Rest],Body):-
searchforbody([predicateRule(X,List)],Rest,Body).

searchforbody([predicateRule(X,List)],[predicate(_,_)|Rest],Bod
y):- searchforbody([predicateRule(X,List)],Rest,Body).

searchforbody([predicateRule(X,List)],[predicateRule(X,Alist)|R
est],Body):-
lengthList(List,Alist,C),
checkflag(C,[predicateRule(X,List)],Rest,Body).

searchforbody([predicateRule(X,List)],[predicateRule(_,_)|Rest]
,Body):- searchforbody([predicateRule(X,List)],Rest,Body).

checkflag(true,_,Rest,Body):- returnbody(Rest,Body).

checkflag(false,Predicate,Rest,Body):-
searchforbody(Predicate,Rest,Body).

returnbody([],[]).
returnbody([body(X,List)|Rest],[body(X,List)|T]):-
returnbody(Rest,T).
returnbody([predicate(_,_)|_],[]).
returnbody([predicateRule(_,_)|_],[]).

unifypredicate(List,List,[],Uterms,Uterms).

unifypredicate(List,Ulist2,[[v,A,B]|C],Uterms,[[o,B,D]|OUterms]
):-
searchforcompeq(A,Uterms,D),
search(B,List,D,Mlist),
unifypredicate(Mlist,Ulist2,C,Uterms,OUterms).

unifypredicate(List,Ulist,[[_,_,_]|C],Uterms,OUterms):-
unifypredicate(List,Ulist,C,Uterms,OUterms).

searchforcompeq(B,[[i,B,C]|_],C).
searchforcompeq(B,[],B).
searchforcompeq(B,[_|T],C):-searchforcompeq(B,T,C).
searchwhengoback(C,[[v,B,C]|_],B,true).
searchwhengoback(B,[],B,false).
searchwhengoback(B,[_|T],C,D):-searchwhengoback(B,T,C,D).
```

```
%%%%% Apply the substitutions of the head to the body %%%%%
unifybody([body(X,List)],[body(X,List)],[]).

unifybody([body(X,List)],[body(X,Ulist)],[[i,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

unifybody([body(X,List)],[body(X,Ulist)],[[o,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

unifybody([body(X,List)],[body(X,Ulist)],[[v,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

unifybody1([body(X,List)],[body(X,List)],[]).

unifybody1([body(X,List)],[body(X,Ulist)],[[i,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

unifybody1([body(X,List)],[body(X,Ulist)],[[o,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

unifybody1([body(X,List)],[body(X,Ulist)],[[v,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

ubody(List,List,[]).

ubody(List,Ulist,[[o,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

ubody(List,Ulist,[[v,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

ubody(List,Ulist,[[i,A,B]|C]):-
search(A,List,B,Mlist),
ubody(Mlist,Ulist,C).

search(_,[],_,[]).

searchi(A,[A],B,B).
searchi([A],[A],B,B).
searchi(_,[B],_,[B]).

search(A,[[predicate(A,C)]|REST],B,[[predicate(B,C)]|UREST]):-
search(A,REST,B,UREST).
```

```
search(A,[[predicate(C,[A])]|REST],B,[[predicate(C,[B])]|UREST]
):-
search(A,REST,B,UREST).

search(A,[[predicate(A,C)|REST]|R],B,[[predicate(B,C)|UREST]|UR
]):-
searchi(A,REST,B,UREST),
search(A,R,B,UR).

search(A,[[predicate(C,[A|R])|RR]|REST],B,[[predicate(C,[B|UR])
|URR]|UREST]):-
search(A,R,B,UR),
searchi(A,RR,B,URR),
search(A,REST,B,UREST).

search(A,[[o,A,C]|REST],B,[[o,B,C]|UREST]):-
search(A,REST,B,UREST).

search(A,[[o,C,A]|REST],B,[[o,C,B]|UREST]):-
search(A,REST,B,UREST).

search([A],[[o,A,C]|REST],[B],[[o,B,C]|UREST]):-
search(A,REST,B,UREST).

search([A],[[o,C,A]|REST],[B],[[o,C,B]|UREST]):-
search(A,REST,B,UREST).

search(A,[[A|C]|REST],B,[[B|C]|UREST]):-search(A,REST,B,UREST).

search(A,[[X|N]|REST],B,[[X|M]|UREST]):-
searchi(A,N,B,M),search(A,REST,B,UREST).

search(A,[[query(A,C)]|REST],B,[[query(B,C)]|UREST]):-
search(A,REST,B,UREST).

search(A,[[query(C,[A|R])]|REST],B,[[query(C,[B|UR])]|UREST]):-
search(A,R,B,UR),
search(A,REST,B,UREST).

search([A],[A|REST],[],[[]|UREST]):-search(A,REST,[],UREST).

search([A],[X|REST],[],[X|UREST]):-search(A,REST,[],UREST).

search(A,[A|REST],B,[B|UREST]):-search(A,REST,B,UREST).
search(A,[X|REST],B,[X|UREST]):-search(A,REST,B,UREST).
search([A,B],Terms,[C|D]):- search(A,Terms,C),
search(B,Terms,D).
search(A,[],A).
search(A,[[i,A,C]|_],C).
search(A,[[o,A,C]|_],C).
search(A,[_|Rest],C):-search(A,Rest,C).
```

```
searchDB(X,[body(_,_)|Rest],Predicate):-
searchDB(X,Rest,Predicate).

searchDB([query(X,List)],[predicate(X,Alist)|Rest],Predicate):-
lengthList(List,Alist,C),
checkflagDB(C,X,List,[predicate(X,Alist)|Rest],Predicate).

searchDB([query(X,List)],[predicateRule(X,Alist)|Rest],Predicat
e):-
lengthList(List,Alist,C),
checkflagDB(C,X,List,[predicateRule(X,Alist)|Rest],Predicate).

searchDB([query(X,List)],[_|Rest],Predicate):-
searchDB([query(X,List)],Rest,Predicate).

checkflagDB(true,X,List,[predicate(X,Alist)|Rest],Predicate):-
checkargs(List,Alist,ArgRes),
checkflagArg(ArgRes,X,List,[predicate(X,Alist)|Rest],Predicate)
.

checkflagDB(true,X,List,[predicateRule(X,Alist)|Rest],Predicate
):-
checkargs(List,Alist,ArgRes),
checkflagArg(ArgRes,X,List,[predicateRule(X,Alist)|Rest],
Predicate).

checkflagDB(false,X,List,[_|Rest],Predicate):
searchDB([query(X,List)],Rest,Predicate).

checkflagArg(true,_,_,[predicate(X,Alist)|_],Predicate):-
valterm([predicate(X,Alist)],Predicate).

checkflagArg(true,_,_,[predicateRule(X,Alist)|_],Predicate):-
valterm([predicateRule(X,Alist)],Predicate).

checkflagArg(false,X,List,[_|Rest],Predicate):-
searchDB([query(X,List)],Rest,Predicate).

insertPredforBT(X,[],[X]).
insertPredforBT(X,BTList,[X|BTList]).


%%%%%%%%%%%%%%%%%%% UNIFICATION PREDICATES %%%%%%%%%%%%%%%%%

unify([query(X,List)],[predicate(X,Alist)],[predicate(X,Ulist)]
,IUterms,OUterms):-ulist(List,Alist,Ulist,IUterms,OUterms).

unify([query(X,List)],[predicateRule(X,Alist)],[predicateRule(X
,Ulist)],IUterms,OUterms):-
ulist(List,Alist,Ulist,IUterms,OUterms).

ulist([],[],[],L,L).
```

```
ulist([],[B],[],L,[[i,B,[]]|L]).
ulist([],B,[],L,[[i,B,[]]|L]).
ulist(_,[],[],L,L).
ulist(B,'_',B,L,L).
ulist([[o,A,B]],[[[o,A1,B1]]],[[o,A,B]],L,[[i,A1,A],[i,B1,B]|L]
).

ulist([[]|List],[[]|Alist],[[]|Ulist],L,L1):-
ulist(List,Alist,Ulist,L,L1).

ulist([[]|List],[A|Alist],[[]|Ulist],L,[[i,A,[]]|L1]):-
ulist(List,Alist,Ulist,L,L1).

ulist([A|List],['_'|Alist],[A|Ulist],L,L1):-
ulist(List,Alist,Ulist,L,L1).

ulist(['_'|List],[_|Alist],['_'|Ulist],L,L1):-
ulist(List,Alist,Ulist,L,L1).

%%%%% occur check %%%%%
ulist([X|List],[functor(Y,Flist)|Alist],[U|Ulist],L,L1):-
occurcheck(X,Flist,Flag),
checkoccurflag(Flag, [X|List],[functor(Y,Flist)|Alist]
,[U|Ulist],L,L1).

%%%%% occur check %%%%%
ulist([X|List],[[A|Rest]|Alist],[U|Ulist],L,L1):-
occurcheck(X,[A|Rest],Flag),
checkoccurflag(Flag, [X|List],[functor(Y,Flist)|Alist]
,[U|Ulist],L,L1).


ulist([[_|Tail1]|List],[['_',Tail2]|Alist],[['_'|Tail]|Ulist],L
,L1):-
ulist(Tail1,Tail2,Tail,L,L2), ulist(List,Alist,Ulist,L2,L1).

ulist([[predicate(A,[Arg])|Tail1]|List],[[predicate('_',[Arg2])
|Tail2]|Alist],[[predicate(A,[Arg3])|Tail]|Ulist],L,L4):-
ulist([Arg],[Arg2],Arg3,L,L1), ulist(Tail1,Tail2,Tail,L1,L2),
ulist(List,Alist,Ulist,L2,L4).

ulist([[predicate(A,[Arg])|Tail1]|List],[[predicate(B,[Arg2])|T
ail2]|Alist],[[predicate(A,[Arg3])|Tail]|Ulist],L,[[v,B,A]|L4])
:-
valterm1(A, AA),
valterm1(B , BB),
valterm(and,AA,BB , true),
ulist(Arg,Arg2,Arg3,L,L1),
ulist(Tail1,Tail2,Tail,L1,L2), ulist(List,Alist,Ulist,L2,L4).
```

```
ulist([[predicate(A,[Arg])|Tail1]|List],[[predicate(B,[Arg2])|T
ail2]|Alist],[[predicate(B,[Arg3])|Tail]|Ulist],L,[[o,A,B]|L4])
:-
valterm1(A, AA),
valterm(hash,AA,false,true),
ulist(Arg,Arg2,Arg3,L,L1),
ulist(Tail1,Tail2,Tail,L1,L2), ulist(List,Alist,Ulist,L2,L4).

ulist([[predicate(A,[Arg])|Tail1]|List],[[predicate(B,[Arg2])|T
ail2]|Alist],[[predicate(A,[Arg3])|Tail]|Ulist],L,L6):-
valterm1(B,BB),
valterm(hash,BB,false,true),
ulist(Arg,Arg2,Arg3,L,L1),
ulist(Tail1,Tail2,Tail,L1,L2),
ulist(List,Alist,Ulist,L2,L4),
checkavailability(B,L4,Flag),checkflag1(Flag,B,A,L4,L6).

ulist([[query(A,[Arg])|Tail1]|List],[[query(B,[Arg2])|Tail2]|Al
ist],[[query(A,[Arg3])|Tail]|Ulist],L,[[v,B,A]|L4]):-
valterm1(A, AA),
valterm1(B , BB),
valterm(and,AA,BB , true),
ulist(Arg,Arg2,Arg3,L,L1),
ulist(Tail1,Tail2,Tail,L1,L2), ulist(List,Alist,Ulist,L2,L4).

ulist([[query(A,[Arg])|Tail1]|List],[[query(B,[Arg2])|Tail2]|Al
ist],[[query(B,[Arg3])|Tail]|Ulist],L,[[o,A,B]|L4]):-
valterm1(A, AA) ,
valterm(hash,AA,false,true),
ulist(Arg,Arg2,Arg3,L,L1),
ulist(Tail1,Tail2,Tail,L1,L2),
ulist(List,Alist,Ulist,L2,L4).

ulist([[query(A,[Arg])|Tail1]|List],[[query(B,[Arg2])|Tail2]|Al
ist],[[query(A,[Arg3])|Tail]|Ulist],L,L6):-
valterm1(B, BB),
valterm(hash,BB,false,true),
ulist(Arg,Arg2,Arg3,L,L1),
ulist(Tail1,Tail2,Tail,L1,L2), ulist(List,Alist,Ulist,L2,L4),
checkavailability(B,L4,Flag),
checkflag1(Flag,B,A,L4,L6).

ulist([[A|Tail1]|List],[[B|Tail2]|Alist],[[A|Tail]|Ulist],L,[[v
,B,A]|L2]):-
valterm1(A, AA),
valterm1(B , BB),
valterm(and,AA,BB , true),
ulist(Tail1,Tail2,Tail,L,L1), ulist(List,Alist,Ulist,L1,L2).

ulist([[A|Tail1]|List],[[B|Tail2]|Alist],[[B|Tail]|Ulist],L,[[o
,A,B]|L2]):-
valterm1(A, AA),
```

```
valterm(hash,AA,false,true), ulist(Tail1,Tail2,Tail,L,L1),
ulist(List,Alist,Ulist,L1,L2).

ulist([[A|Tail1]|List],[[B,Tail2]|Alist],[[A|Tail]|Ulist],L,[[i
,B,A]|L2]):-
valterm1(B, BB),
valterm(hash,BB,false,true), ulist(Tail1,Tail2,Tail,L,L1),
ulist(List,Alist,Ulist,L1,L2).

ulist([[A|Tail1]|List],[[A|Tail2]|Alist],[[A|Tail]|Ulist],L,L2)
:-
ulist(Tail1,Tail2,Tail,L,L1), ulist(List,Alist,Ulist,L1,L2).

ulist([[A|C]|List],[B|Alist],[[A|C]|Ulist],L,[[i,B,[A|C]]|L1]):
-
valterm1(B, BB),
valterm(hash,BB,false,true),
ulist(List,Alist,Ulist,L,L1).

ulist([A|List],[[[o,Q,F]]|Alist],[[o,Q,F]|Ulist],L,[[o,A,[o,Q,F
]]|L1]):-
valterm1(A, AA),
valterm(and,AA,true,true),
ulist(List,Alist,Ulist,L,L1).

ulist([A|List],[B|Alist],[A|Ulist],L,[[v,B,A]|L1]):-
valterm1(A, AA),
valterm1(B , BB),
valterm(and,AA,BB , true),
ulist(List,Alist,Ulist,L,L1).

ulist([A|List],[B|Alist],Ulist,L,L1):-
valterm1(A, AA),
valterm(hash,AA,false,true),
checkavailability(A,L,Flag),
checkflag(Flag,[A|List],[B|Alist],Ulist,L,L1).

ulist([A|List],[B|Alist],[A|Ulist],L,[[i,B,A]|L1]):-
valterm1(B, BB),
valterm(hash,BB,false, true),
ulist(List,Alist,Ulist,L,L1).

ulist([A],[A],[A],L,L).

ulist([A|List],[A|Alist],[A|Ulist],L,L1):
ulist(List,Alist,Ulist,L,L1).

ulist([A|C],B,[A|C],L,[[i,B,[A|C]]|L]).

ulist1([A|List],[B|Alist],[B|Ulist],L,L1):-
valterm1(A, AA),
valterm(hash,AA,false,true),
```

```
checkiando(A,L,Flag),
checkflag(Flag),
ulist(List,Alist,Ulist,L,L1).

ulist1([A|List],[B|Alist],[B|Ulist],L,[[o,A,B]|L1]):-
valterm1(A, AA),
valterm(hash,AA,false,true),
removevar(A,L,LLL),
ulist(List,Alist,Ulist,LLL,L1).

checkflag(false,[A|List],[B|Alist],Ulist,L,L1):-
ulist1([A|List],[B|Alist],Ulist,L,L1).

checkflag(true,[A|List],[B|Alist],[B|Ulist],L,[[o,A,B]|L1]):-
ulist(List,Alist,Ulist,L,L1).

checkflag1(false,_,_,L5,L5).

checkflag1(true,B,A,L5,[[i,B,A]|L5]).

occurcheck(_,[],true).
occurcheck(A,[[_,A,_]|_],false).
occurcheck(A,[_|Rest],Flag):-checkavailability(A,Rest,Flag).

checkoccurflag(false,[_|List],[_|Alist],Ulist,L,[fail|L1]):-
ulist(List,Alist,Ulist,L,L1).

checkoccurflag(true,[A|List],[B|Alist],[B|Ulist],L,[[o,A,B]|L1]
):- ulist(List,Alist,Ulist,L,L1).

removevar(_,[],[]).
removevar(A,[[v,A,_]|L],LLL):- removevar(A,L,LLL).
removevar(A,[[X,Y,Z]|L],[[X,Y,Z]|LLL]):- removevar(A,L,LLL).

checkiando(_,[],false).
checkiando(A,[[i,A,_]|_],true).
checkiando(A,[[o,A,_]|_],true).
checkiando(A,[[_,_,_]|L],Flag):- checkiando(A,L,Flag).

checkavailability(_,[],true).
checkavailability(A,[[_,A,_]|_],false).
checkavailability(A,[_|Rest],Flag):-
checkavailability(A,Rest,Flag).

%%%%%%%%%%%%%%%% END OF UNIFICATION PREDICATES %%%%%%%%%%%%%%%%

checkforvar([predicate(_,List)],Flag):- check(List,Flag).

checkforvar([predicateRule(_,List)],Flag):- check(List,Flag).

check([],false).
check([true],false).
```

```
check([[]|B],Flag):-check(B,Flag).
check([[A]|B],Flag):-valterm1(A,false),check(B,Flag).

check([[A|C]|B],Flag):-
valterm1(A,false),check(C,Flag1),check(Flag1,B,Flag).

check([[A|_]|_],Flag):-valterm1(A,true),setFlag(true,Flag).

check([A|_],Flag):-valterm1(A,true),setFlag(true,Flag).
check([A|B],Flag):-valterm1(A,false),check(B,Flag).
check(true,_,Flag):- setFlag(true,Flag).
check(false,B,Flag):- check(B,Flag).

setFlag(true,true).

displayResult1([],Terms,Var):- displayResult(Terms,Var).
displayResult1([fail|_],_,_):- write('false').  %OCCUR CHECK
displayResult1([_|A],Terms,Var):-displayResult1(A,Terms,Var).

displayResult([],_):- write(true).
displayResult([[i,_,_]|T],A):- displayResult(T,A).

displayResult([[o,A,B]|T],A):-
write(A),
write('='),
write(B),
displayResult(T,A).

displayResult([[v,A,B]|T],A):-
write(A),
write('='),
write(B),
displayResult(T,A).

displayResult([[o,_,_]|T],A):- displayResult(T,A).
displayResult([[v,_,_]|T],A):- displayResult(T,A).

checkargs([],[],true).
checkargs(_,['_'],true).
checkargs([[_|_]|_],[[]|_],false).
checkargs([[]|T],[[]|TT],Result):- checkargs(T,TT,Result).

checkargs([[A|B]|T],[[C|D]|TT],Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,G),
checkargs(B,D,F),
valterm(and,F,G,H), checkingVarExistance(H,A1,C1,T,TT,Result).

checkargs([_|T],['_'|TT],Result):-
checkingVarExistance(true,_,_,T,TT,Result).
```

```
checkargs([A|T],[[C]|TT],Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,G), checkingVarExistance(G,A1,C1,T,TT,Result).

checkargs([[A]|T],[C|TT],Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,G), checkingVarExistance(G,A1,C1,T,TT,Result).

checkargs([A|T],[[C|_]|TT],Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,G), checkingVarExistance(G,A1,C1,T,TT,Result).

checkargs([[A|_]|T],[C|TT],Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,G), checkingVarExistance(G,A1,C1,T,TT,Result).

checkargs([A1|T],[A2|TT],Result):-
valterm1(A1 , B),
valterm1(A2 , C),
valterm(or,B,C,D), checkingVarExistance(D,A1,A2,T,TT,Result).

checkargs([A],C,Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,Result).

checkargs(A,C,Result):-
valterm1(A , A1),
valterm1(C , C1),
valterm(or,A1,C1,Result).

checkingVarExistance(true,_,_,LIST1,LIST2,Result):-
checkargs(LIST1,LIST2,Result).

checkingVarExistance(false,A,A,LIST1,LIST2,Result):-
checkargs(LIST1,LIST2,Result).

checkingVarExistance(false,_,_,_,_,Result):-
valterm(false,Result).

valterm1([],false).

valterm1([[_,_,[_,A,_]]],D):-
string_to_list(A,B),
checkvar(B,D).

valterm1([_,_,[_,A,_]],D):-
string_to_list(A,B),
```

```
checkvar(B,D).

valterm1([query(A,_)],D):-
string_to_list(A,B),
checkvar(B,D).

valterm1(query(A,_),D):-
string_to_list(A,B),
checkvar(B,D).

valterm1(predicate(A,_),D):-
string_to_list(A,B),
checkvar(B,D).

valterm1([predicate(A,_)],D):-
string_to_list(A,B),
checkvar(B,D).

valterm1(predicateRule(A,_),D):-
string_to_list(A,B),
checkvar(B,D).

valterm1(body(A,_),D):-
string_to_list(A,B),
checkvar(B,D).

valterm1([A],D):- string_to_list(A,B),checkvar(B,D).
valterm1([A,_],D):- string_to_list(A,B),checkvar(B,D).
valterm1([_,A,_],D):- string_to_list(A,B),checkvar(B,D).

valterm1([[_,A,_]],D):-
string_to_list(A,B),
checkvar(B,D).

valterm1(A,D):-
string_to_list(A,B),
checkvar(B,D).

valterm(hash,true,false,true).
valterm(or,false,false,false).
valterm(or,A,B, true):- bool(A),bool(B).
valterm(and,true,true, true).
valterm(and,A,B, false) :-  bool(A),bool(B).
valterm(false,false).
valterm([predicate(X,List)],[predicate(X,List)]).
valterm([predicateRule(X,List)],[predicateRule(X,List)]).
valterm([body(X,List)],[body(X,List)]).

bool(true).
bool(false).

checkvar([65|_],true).
```

```
checkvar([66|_],true).
checkvar([67|_],true).
checkvar([68|_],true).
checkvar([69|_],true).
checkvar([70|_],true).
checkvar([71|_],true).
checkvar([72|_],true).
checkvar([73|_],true).
checkvar([74|_],true).
checkvar([75|_],true).
checkvar([76|_],true).
checkvar([77|_],true).
checkvar([78|_],true).
checkvar([79|_],true).
checkvar([80|_],true).
checkvar([81|_],true).
checkvar([82|_],true).
checkvar([83|_],true).
checkvar([84|_],true).
checkvar([85|_],true).
checkvar([86|_],true).
checkvar([87|_],true).
checkvar([88|_],true).
checkvar([89|_],true).
checkvar([90|_],true).
checkvar([_|_],false).

finalterm(X,X).

checkflag(true).

lengthList([],[],true).
lengthList([],_,false).
lengthList(_,[],false).
lengthList([_|T1],[_|T2],Result) :- lengthList(T1,T2,Result).
```

APPENDIX B: SCANNER AND PARSER

```
%%%%%%%%%%%%%%%%%%% SCANNER %%%%%%%%%%%%%%%%%%%

interpret(File,Terms):-
see(File),
read_file_to_codes(File,L,[type(text)]),
tokenize(L,Out),
p(Out,_,[],Tree1,Query),
recognizeRule(Tree1,Tree),
sem(Query,Tree,[],_,[],Terms).

tokenize([],[]).

tokenize([46|Rest],[Fullstop|Out]):-   %Full stop
name(Fullstop,[46]),
tokenize(Rest,Out).

tokenize([10|Rest],Out):-    %ignore New line
tokenize(Rest,Out).

tokenize([9|Rest],Out):-    %ignore horizontal tab
tokenize(Rest,Out).

tokenize([32|Rest],Out):-    %ignore space
tokenize(Rest,Out).

tokenize([39|Rest],Out):-    %ignore '
tokenize(Rest,Out).

tokenize([58|Rest],[Colon|Out]):-      %Colon
name(Colon,[58]),
tokenize(Rest,Out).

tokenize([45|Rest],[Hyphen|Out]):-     %Hyphen
name(Hyphen,[45]),
tokenize(Rest,Out).

tokenize([33|Rest],[Cut|Out]):-         %Cut Operator
name(Cut,[33]),
tokenize(Rest,Out).

tokenize([91|Rest],[Llist|Out]):-   %left square bracket
name(Llist,[91]),
tokenize(Rest,Out).
tokenize([93|Rest],[Rlist|Out]):-  %right square bracket
name(Rlist,[93]),
tokenize(Rest,Out).
```

```
tokenize([63|Rest],[Qmark|Out]):-     %Question mark
name(Qmark,[63]),
tokenize(Rest,Out).

tokenize([124|Rest],[Listop|Out]):-      %list operator
name(Listop,[124]),
tokenize(Rest,Out).

tokenize([44|Rest],[Comma|Out]):-      %Comma
name(Comma,[44]),
tokenize(Rest,Out).

tokenize([59|Rest],[DotComma|Out]):-      %Dotted Comma
name(DotComma,[59]),
tokenize(Rest,Out).

tokenize([95|Rest],[Underscore|Out]):-      %underscore
name(Underscore,[95]),
tokenize(Rest,Out).

tokenize([41|Rest],[Rparen|Out]):-      %right paran
name(Rparen,[41]),
tokenize(Rest,Out).

tokenize([40|Rest],[Rparen|Out]):-   %left paran
name(Rparen,[40]),
tokenize(Rest,Out).

tokenize(L,[Word|X]):-tokeniz(L,Word,X).
tokeniz(L,Word,[X|Out]):-
checkempty(L,[],false),
tokenize(L,Rest,WordChs,X),
name(Word,WordChs),
tokenize(Rest,Out).

tokenize([],[],[],_).
tokenize([40|T],T,[],Lparen):-name(Lparen,[40]).
tokenize([44|T],T,[],Comma):- name(Comma,[44]).
tokenize([41|T],T,[],Rparen):- name(Rparen,[41]).
tokenize([93|T],T,[],Rlist):-name(Rlist,[93]).
tokenize([45|T],T,[],Hyphen):-name(Hyphen,[45]).
tokenize([59|T],T,[],DotComma):-name(DotComma,[59]).
tokenize([124|T],T,[],ListOP):-name(ListOP,[124]).
tokenize([39|T],Rest,List,X):-tokenize(T,Rest,List,X).
tokenize([32|T],Rest,List,X):-tokenize(T,Rest,List,X).
tokenize([H|T],Rest,[H|List],X):-tokenize(T,Rest,List,X).

checkempty([],[],true).
checkempty(_,[],false).
```

```
%%%%%%%%%%%%%%%%%% PARSER %%%%%%%%%%%%%%%%%%

p([Id|Tokens],Parselist,[],Oparsetree,Query):-
rl([Id|Tokens],Listsofar,[],Parsetree),
rev(Parsetree,Oparsetree),
q(Listsofar,Parselist,[],RQuery),
rev(RQuery,Query).

rl([Id|Tokens],L,Iparsetree,Oparsetree):-
r([Id|Tokens],L1,Iparsetree,Parsetree), matchperiod(L1,L2),
x(L2,L,Parsetree,Oparsetree).

matchperiod(['.'|Tokens],Tokens).

r([Id|Tokens],L,Iparsetree,Oparsetree):-
atom([Id|Tokens],L1,Iparsetree,Parsetree),
y(L1,L,Parsetree,Oparsetree).

x(['-'|Tokens],['-'|Tokens],Parsetree,Parsetree).

x(['?'|Tokens],['?'|L],Parsetree,Oparsetree):-
x(Tokens,L,Parsetree,Oparsetree).
x([':'|Tokens],[':'|L],Parsetree,Oparsetree):-
x(Tokens,L,Parsetree,Oparsetree).

x([Id|Tokens],L,Iparsetree,Oparsetree):-
rl([Id|Tokens],L,Iparsetree,Oparsetree).

y(['.'|Tokens],['.'|Tokens],Parsetree,Parsetree).
y(['-'|Tokens],Tokens,Parsetree,Parsetree).

y([':'|Tokens],L,Iparsetree,Oparsetree):-
y(Tokens,L1,Iparsetree,Parsetree),al(L1,L,Parsetree,Oparsetree)
.

al([Id|Tokens],L,Iparsetree,Oparsetree):-
atombody([Id|Tokens],L1,Iparsetree,Parsetree),
z(L1,L,Parsetree,Oparsetree).

z(['.'|Tokens],['.'|Tokens],Parsetree,Parsetree).
z([EOF|Tokens],[EOF|Tokens],Parsetree,Parsetree).
z([','|Tokens],L,Iparsetree,Oparsetree):-
al(Tokens,L,Iparsetree,Oparsetree).

atom([Id|Tokens],L,Iparsetree,Oparsetree):-
pred([Id|Tokens],L1,Iparsetree,Parsetree),
matchleftparen(L1,L2),
tl(L2,L3,Parsetree,Oparsetree),
matchrightparen(L3,L).

atombody([Id|Tokens],L,Iparsetree,Oparsetree):-
predbody([Id|Tokens],L1,Iparsetree,Parsetree),
```

```
matchleftparen(L1,L2),
tl(L2,L3,Parsetree,Oparsetree),
matchrightparen(L3,L).

atomquery([Id|Tokens],L,Iparsetree,Oparsetree):-
predquery([Id|Tokens],L1,Iparsetree,Parsetree),
matchleftparen(L1,L2),
tl(L2,L3,Parsetree,Oparsetree),
matchrightparen(L3,L).

matchleftparen(['('|Tokens],Tokens).
matchrightparen([')'|Tokens],Tokens).

tl([Id|Tokens],L,Iparsetree,Oparsetree):-
t([Id|Tokens],L1,Iparsetree,Parsetree),
m(L1,L,Parsetree,Oparsetree).

tl(['['|Tokens],L,Iparsetree,Oparsetree):-
t(['['|Tokens],L1,Iparsetree,Parsetree),
m(L1,L,Parsetree,Oparsetree).

fl([Id|Tokens],L,Iparsetree,Functor):-
ft([Id|Tokens],L1,Iparsetree,Func),
fm(L1,L,Func,Functor).

fl(['['|Tokens],L,Iparsetree,Functor):-
ft(['['|Tokens],L1,Iparsetree,Func),
fm(L1,L,Func,Functor).

ft(['['|Tokens],L,Iparsetree,Functor):-
ll(Tokens,L1,[],Arglist),
inserttofunctor(Arglist,Iparsetree,Functor),
matchrightbraket(L1,L).

ft(['['|Tokens],L,Iparsetree,Functor):-
listHT(Tokens,L1,[],Arglist),
inserttofunctor(Arglist,Iparsetree,Functor),
matchrightbraket(L1,L).

ft([Id|Tokens],L,Iparsetree,Functor):-
inserttofunctor(Id,Iparsetree,Func),
n(Tokens,L,Func,Functor).

fm([')'|Tokens],[')'|Tokens],predicate(Name,Arg),predicate(Name
,Rarg)):-
rev(Arg,Rarg).

fm([')'|Tokens],[')'|Tokens],predicateRule(Name,Arg),predicateR
ule(Name,Rarg)) :-
 rev(Arg,Rarg).

fm([')'|Tokens],[')'|Tokens],body(Name,Arg),body(Name,Rarg)):-
```

```
rev(Arg,Rarg).

fm([')'|Tokens],[')'|Tokens],query(Name,Arg),query(Name,Rarg)):
-
rev(Arg,Rarg).

fm([')'|Tokens],[')'|Tokens],[body(Name,Arg)|_],[body(Name,Rarg
)|_]):-
rev(Arg,Rarg).

fm([')'|Tokens],[')'|Tokens],[predicate(Name,Arg)|_],[predicate
(Name,Rarg)|_]):-
rev(Arg,Rarg).

fm([')'|Tokens],[')'|Tokens],[predicateRule(Name,Arg)|_],[predi
cateRule(Name,Rarg)|_]) :-
rev(Arg,Rarg).

fm([','|Tokens],L,Ifunctor,Ofunctor):
fl(Tokens,L,Ifunctor,Ofunctor).

ll(['['|Tokens],L,Iparsetree,Oparsetree):-
tt(['['|Tokens],L1,Iparsetree,Parsetree),
mm(L1,L,Parsetree,Oparsetree).

ll([']'|Tokens],[']'|Tokens],Iparsetree,Iparsetree).

ll([Id|Tokens],L,Ilist,Olist):- tt([Id|Tokens],L1,Ilist,List),
mm(L1,L,List,Olist).

listHT([Id|Tokens],L,Ilist,Olist):-
tt([Id|Tokens],L1,Ilist,List),
mmm(L1,L,List,Olist).

m([')'|Tokens],[')'|Tokens],[predicate(Name,Arg)|L],[predicate(
Name,Rarg)|L]):-
rev(Arg,Rarg).

m([')'|Tokens],[')'|Tokens],[body(Name,Arg)|L],[body(Name,Rarg)
|L]):-
rev(Arg,Rarg).

m([')'|Tokens],[')'|Tokens],[query(Name,Arg)|L],[query(Name,Rar
g)|L]):-
rev(Arg,Rarg).

m([','|Tokens],L,Iparsetree,Oparsetree):-
tl(Tokens,L,Iparsetree,Oparsetree).

mm([']'|Tokens],[']'|Tokens],List,Rlist):-rev(List,Rlist).

mm([','|Tokens],L,Ilist,Olist):- ll(Tokens,L,Ilist,Olist).
```

```
mmm(['|'|Tokens],L,Ilist,Olist):- lll(Tokens,L,Ilist,Olist).

lll([Id|Tokens],L,Ilist,Olist):- inserlistArg(Id,Ilist,List),
mm(Tokens,L,List,Olist).

t(['['|Tokens],L,Iparsetree,Oparsetree):-
ll(Tokens,L1,[],Arglist),
insertlist(Arglist,Iparsetree,Oparsetree),
matchrightbraket(L1,L).

t(['['|Tokens],L,Iparsetree,Oparsetree):-
listHT(Tokens,L1,[],Arglist),
insertlist(Arglist,Iparsetree,Oparsetree),
matchrightbraket(L1,L).


t([Id|Tokens],L,Iparsetree,Oparsetree):-
insertargument(Id,Iparsetree,Parsetree),
n(Tokens,L,Parsetree,Oparsetree).

tt(['['|Tokens],L,Ilist,Olist):- ll(Tokens,L1,[],Arglist),
insertlist(Arglist,Ilist,Olist),
matchrightbraket(L1,L).

tt(['['|Tokens],L,Ilist,Olist):- listHT(Tokens,L1,[],Arglist),
insertlist(Arglist,Ilist,Olist),
matchrightbraket(L1,L).

tt([Id|Tokens],L,Ilist,Olist):- inserlistArg(Id,Ilist,List),
n(Tokens,L,List,Olist).

n([')'|Tokens],[')'|Tokens],Parsetree,Parsetree).
n(['|'|Tokens],['|'|Tokens],Parsetree,Parsetree).
n([']'|Tokens],[']'|Tokens],Parsetree,Parsetree).
n([','|Tokens],[','|Tokens],Parsetree,Parsetree).

n(['('|Tokens],L,Iparsetree,Oparsetree):-
fl(Tokens,L1,Iparsetree,Functor),
insertfunctor(Functor,Iparsetree,Oparsetree),
matchrightparen(L1,L).

matchrightbraket([']'|Tokens],Tokens).

pred([Id|Tokens],Tokens,Iparsetree,Oparsetree):-
insertpredicate(Id,Iparsetree,Oparsetree).

predbody([Id|Tokens],Tokens,Iparsetree,Oparsetree):-
insertbody(Id,Iparsetree,Oparsetree).

predquery([Id|Tokens],Tokens,Iparsetree,Oparsetree):-
insertquery(Id,Iparsetree,Oparsetree).
```

```
q(['-'|Tokens],Tokens,Query,Query).

q([':'|Tokens],L,Iq,Ouery):-
q(Tokens,L2,Iq,IIq),
ql(L2,L1,IIq,Ouery),
matchperiod(L1,L).


q(['?'|Tokens],L,Iq,Ouery):-
q(Tokens,L2,Iq,IIq),
ql(L2,L1,IIq,Ouery),
matchperiod(L1,L).

insertargument(Id,[predicate(Name,Arg)|L],[predicate(Name,[Id|A
rg])|L]).

insertargument(Id,[body(Name,Arg)|L],[body(Name,[Id|Arg])|L]).

insertargument(Id,[query(Name,Arg)|L],[query(Name,[Id|Arg])|L])
.

ql([Id|Tokens],L,IQ,QUERY):- atomquery([Id|Tokens],L1,IQ,Q),
f(L1,L,Q,QUERY).

f(['.'|Tokens],['.'|Tokens],Parsetree,Parsetree).
f([EOF|Tokens],[EOF|Tokens],Parsetree,Parsetree).

f([','|Tokens],L,Iparsetree,Oparsetree):
ql(Tokens,L,Iparsetree,Oparsetree).

insertpredicate(Id,[],[predicate(Id,[])]).
insertpredicate(Id,L,[predicate(Id,[])|L]).

insertbody(Id,L,[body(Id,[])|L]).

insertquery(Id,[],[query(Id,[])]).
insertquery(Id,L,[query(Id,[])|L]).

insertlist(List,[predicate(Name,[])|L],[predicate(Name,[List])|
L]).

insertlist(List,[body(Name,[])|L],[body(Name,[List])|L]).

insertlist(List,[query(Name,[])|L],[query(Name,[List])|L]).

insertlist(List,[predicate(Name,Arg)|L],[predicate(Name,[List|A
rg])|L]).

insertlist(List,[body(Name,Arg)|L],[body(Name,[List|Arg])|L]).
```

```
insertlist(List,[query(Name,Arg)|L],[query(Name,[List|Arg])|L])
.

insertlist(INList,List,[INList|List]).
inserlistArg(Id,[],[Id]).

inserlistArg(Id,Arg,[Id|Arg]).

inserttofunctor(Id,[predicate(_,[Funcname|_])|_],functor(Funcna
me,[Id])).

inserttofunctor(Id,[body(_,[Funcname|_])|_],functor(Funcname,[I
d])).

inserttofunctor(Id,[query(_,[Funcname|_])|_],functor(Funcname,[
Id])).

inserttofunctor(Id,functor(Funcname,FuncArg),functor(Funcname,[
Id|FuncArg])).

inserttofunctor(Id,[predicate],predicate(Id)).
inserttofunctor(Id,[body],body(Id)).
inserttofunctor(Id,[body|A],[body(Id)|A]).
inserttofunctor(Id,[predicateRule],predicateRule(Id)).
inserttofunctor(Id,[query],query(Id)).
inserttofunctor(Id,[Funcname],functor(Funcname,[Id])).
inserttofunctor(Id,predicate(X),predicate(X,[Id])).
inserttofunctor(Id,body(X),body(X,[Id])).
inserttofunctor(Id,[body(X)|A],[body(X,[Id])|A]).
inserttofunctor(Id,query(X),query(X,[Id])).
inserttofunctor(Id,predicateRule(X),predicateRule(X,[Id])).
inserttofunctor(Id,predicate(X,Y),predicate(X,[Id|Y])).
inserttofunctor(Id,body(X,Y),body(X,[Id|Y])).
inserttofunctor(Id,predicateRule(X,Y),predicateRule(X,[Id|Y])).
inserttofunctor(Id,query(X,Y),query(X,[Id|Y])).
insertfunctor(Functor,[predicate(Name,[_|Arg])|L],[predicate(Na
me,[Functor|Arg])|L]).
insertfunctor(Functor,[body(Name,[_|Arg])|L],[body(Name,[Functo
r|Arg])|L]).
insertfunctor(Functor,[query(Name,[_|Arg])|L],[query(Name,[Func
tor|Arg])|L]).
insertfunctor(functor(Name,[Arg]),[Name],[functor(Name,[Arg])])
.
insertfunctor(functor(Name,Arg),[Name],[functor(Name,Arg)]).
insertfunctor(predicate(Name,[Arg]),[predicate],[predicate(Name
,[Arg])]).

insertfunctor(predicate(Name,Arg),[predicate],[predicate(Name,A
rg)]).

insertfunctor(predicateRule(Name,[Arg]),[predicateRule],[predic
ateRule(Name,[Arg])]).
```

```
insertfunctor(predicateRule(Name,Arg),[predicateRule],[predicat
eRule(Name,Arg)]).

insertfunctor(body(Name,[Arg]),[body],[body(Name,[Arg])]).

insertfunctor(body(Name,Arg),[body],[body(Name,Arg)]).

insertfunctor(query(Name,[Arg]),[query],[query(Name,[Arg])]).

insertfunctor(query(Name,Arg),[query],[query(Name,Arg)]).

recognizeRule([],[]).

recognizeRule([predicate(X,List)|Tail1],[predicateRule(X,List)|
Tail2]):-
recognize(Tail1,Tail2).

recognize([body(X,List)|Tail1],[body(X,List)|Tail2]):-
recognizeRule(Tail1,Tail2).

recognizeRule([X|Tail1],[X|Tail2]):-recognizeRule(Tail1,Tail2).

rev([],X,X).
rev([X|Y],Z,W) :- rev(Y,[X|Z],W).
```

APPENDIX C: PREDICATE SETS

The predicate sets for every non-terminal in the Prolog grammar (after removing the left-factoring and left-recursion) are:

Grammar	Predicate Sets
PROGRAM → RULELIST QUERY	{id}
RULELIST → RULE **.** X	{id}
X → RULELIST	{id}
X→ ε	{:, ?}
RULE→ ATOM Y	{id}
Y → **:-** ARGUMENTLIST	{:}
Y→ ε	{.}
ARGUMENTLIST→ATOM Z	{id}
Z→ **,** ARGUMENTLIST	{,}
Z→ ε	{.}
ATOM → PRED (PREDLIST)	{id}
PREDLIST → T M	{id, [}
M→ **,** PREDLIST	{,}
M→ ε	{),]}
T → id N	{id}
T→[LL]	{[}
N→ ε	{‘,’ ,),]}
N→(PREDLIST)	{(}
PRED→id	{id}
LL→ ε	{]}
LL→TT MM	{ [, id}
MM→ ε	{] }
MM→**,** LL	{ , }
MM→ | LLL	{ | }
TT→[K	{[}
TT→id	{id}
K→]	{]}
K→LL]	{ [, id}
LLL→ [ ]	{[}
LLL→ id | LLL	{id}
QUERY → **:-** BODYLIST	{:}
QUERY → **?-** BODYLIST	{?}

BIBLIOGRAPHY

[1] Abelson H. and Sussman G. J., with Sussman J., "*Structure and Interpretation of Computer Programs",* MIT PRESS, 1996

[2] Allison L., "An Executable Prolog Semantics", *Computer History Museum Mountain View, CA, United States, ALGOL Bulletin*, Issue 50, Dec. 1983, pp.10-18

[3] Almeida J. B., Frade M. J., Pinto J. S. and Melo de Sousa S.," *Rigorous Software Development: An Introduction to Program Verification",* Springer,1st edition, 2011

[4] Barbuti R, De Francesco N., Mancarella P. and Santone A." Towards a logical semantics for pure Prolog", *Science of Computer Programming*, Volume 32, Issues 1-3, September 1998, Pages 145-176

[5] Bossi A., Bugliesi M. and Fabris M., "A new fixpoint semantics for Prolog", *in: Proc. of the 10th Intemat. Conf. on Logic Programming, MIT Press, Cambridge, MA*, 1993, pp. 374-389.

[6]Bowen K.A., Kowalski R.A.: "Amalgamating language and metalanguage in logic programming", *In Logic Programming, Academic Press*, London 1982 153-172

[7] Chalier B. L., Rossi S. and Hentenryck P. V. "An Abstract Interpretation Framework which Accurately Handles PROLOG Search Rule and the Cut". In M.Bruynooghe, editor, *Proceedings of the 1994 Int'l Symposium on Logic Programming*, pages 157-171, MIT Press, 1994.

[8] Chang C. and Lee R., "*Symbolic logic and mechanical theorem proving",* academic *press*, 1974

[9] Christiansen H., "Teaching Computer Languages and Elementary Theory for Mixed Audiences at University Level", *Computer Science Education journal*, vol. 14, Issue 3, p.205-234

[10] Codish M. and Demoen B., Analysing logic programs using "Prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.

[11] Costantini S., "Meta-reasoning: a Survey", *Computational Logic: Logic Programming and Beyond – Essays in Honour of Robert A. Kowalski* (LNAI Volumes 2408), Springer-Verlag, 2002, pp 253-288

[12] Covington M. A.,"Tokenization using DCG Rules", *Artificial Intelligence*, 2000, Pages: 1-9

[13] Debray S. K. and Mishra P., "Denotational and operational semantics for Prolog", *The Journal of Logic Programming*, Volume 5, Issue 1, March 1988, Pages 61-91

[14] Dˇzeroski S., Cussens J. and Manandhar S. , "An introduction to inductive logic programming and learning language in logic". *In Learning language in logic, Springer-Verlag New York, Inc.*, 2001, pp 3-35.

[15] Emden M. van and Kowalski R, "The semantics of logic as a programming language". *Journal of the ACM*, 23:733–742, 1976.

[16] Endriss U., "An Introduction to Prolog Programming", *Institute for Logic, Language and Computation*, Language and Computation (ILLC) at the University of Amsterdam, 2007.

[17] Falaschi M., Levi G., Gabbrielli M., and Palamidessi C., "A new declarative semantics for logic languages". In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 993–1005. MIT Press, 1988.

[18] Gallier J. H.,"*Logic for Computer Science: Foundations of Automatic Theorem Proving"*. Harper & Row Publishers, 1985, pp.146-255

[19] Gallier J.H., "*Logic for Computer Science: Foundation of Automatic Theorem Proving"*, John Wiley, 1987

[20] Gruenstein A., "Prolog in Java", Linguistics project in Stanford University.

[21] Gupta P., "The Design and Implementation of a Prolog Parser Using JAVACC", master thesis, *University of North Texas*, August 2002

[22] Hill P.M. and Lloyd J.W., "Analysis of Meta programs". In Abramson, H., Rogers, M.H., eds.: *Meta-Programming in Logic Programming*, Cambridge, Mass., THE MIT Press (1988) 23-51

[23] Ivan B., "*Prolog Programming for Artificial Intelligence"*, 3rd Edition, Pearson Education, 2007.

[24] James Lu and Jerud J. Mead Prolog – "A Tutorial Introduction, Journal of Microcomputer Applications", Volume: 5, Issue: 1, Publisher: Naturalia Monspeliensia, 1982, Pages: 1-40

[25] Jones N. D. and Mycroft A., "Stepwise Development of Operational and Denotational Semantics for PROLOG", in *Proc. 1984 Int. Symposium on Logic Programming*, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, 289-298.

[26] Kowalski, R. “Algorithm = logic + control”. *Commun. ACM 22*, 7 (July 1979), 424-436.

[27] Lloyd J. W., “*Foundations of Logic Programming”*, 2nd Edition, Springer 1987

[28] Louden K. C., “Compiler Construction: Principles and Practice”, PWS Publishing Company, 1997

[29] Luger G. F. **and** Stubblefield W. A., “*AI Algorithms, Data Structures, and Idioms in PROLOG, LISP, and Java”*, Addison-Wesley, Aug 2008

[30] Martelli A. and Montanari U., “An Efficient Unification Algorithm”. *ACM Trans. Program. Lang. Syst.* 4, 2 April 1982, 258-282.

[31] Mazonka O. and Cristofani D. B., “A Very Short Self-Interpreter”, *The Computing Research Repository CoRR* , cs. PL/0311032,2003

[32] McCarthy, J.e.a.,”*The LISP 1.5 Programmer's Manual*”, MIT Press, August 1962

[33] Mogensen T., Schmidt D and Sudborough I. H. (Eds.), “*The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones”*, Springer, March 10, 2003

[34] Moore, R. C., “The Role of Logic in Artificial Intelligence”, *Center for the Study of Language and Information*, issue 33, 1985

[35] Moore, R. C. “Reasoning from Incomplete Knowledge in a Procedural Deduction System”, *Garland Publishing, Inc*., New York, New York, 1980,p.28.

[36] Nicholson T. and Foo N., “A denotational semantics for Prolog”. *ACM Trans. Program. Lang. Syst.* 11, 4 October 1989, 650-665

[37] Nilson U. and Maluszynski J.,” *Logic, Programming and Prolog”*, John Wiley and Sons, 1999

[38] Robinson J. A., “A Machine-Oriented Logic Based on the Resolution Principle”. *J. ACM* 12, 1 January 1965, 23-41

[39] Slonneger K. and Kurtz B. L., “*Formal Syntax and Semantics of Programming Languages”*, Addison-Wesley, 1995.

[40] Smith, B.C.,” Reflection and semantics in Lisp,” Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '84), ACM , 1984, pp. 23-35

[41] Spoto F., “Operational and Goal-Independent Denotational Semantics for Prolog with Cut”, *Journal of Logic Programming*, volume 42, no. 1, 2000, pp. 1-46

[42] Tamir D. E. and Kandel A. ,”Logic Programming and the Execution Model of Prolog”, *Information Sciences - Applications*, Volume 4, Issue 3, November 1995, Pages 167-191

[43] Warren D. H. D., Pereira L. M., and Pereira F., “Prolog - The Language and Its Implementation Compared with Lisp”. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. ACM, 1977, pp 109-115.

[44] Covington M.A., Nute D. and Vellino A., ”*Prolog Programming in Depth*”, Prentice Hall, 1995