|  |
| --- |
| CSC 330 Test 3 |
| Team D |
| **Samuel Willis, David Bell, Philip Snead, Jordan Casoli, Warren Spencer, Robert Martin, Keiran Reilly, Kevin Dahl, Lucas Main** |

11/27/2013

Contents

[History of Prolog 2](#_Toc373350396)

[Unification and Instantiation 4](#_Toc373350397)

[Datalog: Prolog for Databases 5](#_Toc373350398)

[Backtracking and Non Determinism 6](#_Toc373350399)

[References: 9](#_Toc373350400)

# History of Prolog

Prolog, which stands for “PROgrammation en LOGique” or “Programming in Logic” was created in 1972 in Marseilles, France. Prolog was the end result of a project which aimed to combine logical deductions with natural language processing (NLP). Initially, the combination of logical programming and NLP was attempted using Algol-W, which allowed a very limited form of NLP communication with the computer.

The first Prolog system was created by Philippe Roussel in Algol-W. At the same time, Alain Colmerauer and Robert Pasero created a man-machine communication system. This communication system was the first Prolog program, which was written at the same time as the Prolog language. The system knew about pronouns, articles, prepositions, proper nouns (based on a required asterisk marker), and common nouns, all based in the French language.

The use of logic programming was to represent grammar and use resolution to parse sentences. A large natural language processing system was the first major program written in Prolog. While language was the first concept realized by Prolog, people were more interested in using logic, and programming languages like Prolog, to represent AI systems.

Using the initial version of Prolog, a general problem-solving system called Sugiton and a symbolic computation system were developed. The continued use and experimentation with the initial version of Prolog gave way to the final version of Prolog, a standalone language. Over the next couple years, additional features and improvements were added to the increasingly popular Prolog language.

Recent applications of Prolog, in areas other than databases, have been in expert systems. Expert systems are computer systems that imitate the decision making of human experts. A particularly interesting example of Prolog as an expert system is pigE. PigE is the back-end to DOS based mathematical modelling package AUSPIG. AUSPIG simulates pig growth and reproduction, providing insight to into a pig’s economic and biological status throughout the pig’s life. This system can also be used to experiment with different ways of increasing a pig farms profitability and, in certain cases, it has been hugely successful. However, AUSPIG required an expert to maximize the utility of the output due to its complexity, which was often large arrays of numbers requiring multiple screens to read. This lead to the development of pigE, which is an intelligent post-processor that allows the user to bypass the complex output and access a single screen showing any biological inefficiencies as well as managerial strategy recommendations. PigE takes the AUSPIG report and converts it into various Prolog facts, it then prompts the user for certain information. It then processes these rules, along with the user information, and outputs the recommendations. It was found that pigE outperformed human experts, and was considered a success due to Prologs adaptability and extendibility.

Another interesting recent application of Prolog comes in the form of a machine developed by IBM called Watson. Watson is an artificially intelligent machine that has the ability to answer questions posed in natural language. It does this by analyzing a question and producing hypotheses, collecting evidence for said hypotheses, and presenting its results along with scored levels of confidence. Watson was originally intended to compete on the TV show Jeopardy! and it did so successfully. Since then IBM has found applications for Watson in both the financial and medical fields, exploiting its efficiency in storing, retrieving and analyzing data by using it to provide recommendations to experts using up-to-date statistics and information.

# Unification and Instantiation

Prolog implements unification by following a series of simple rules. Formally defined, they are as follows:

1. If term1 and term2 are constants, then term1 and term2 unify if and only if they are the same atom, or the same number.
2. If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2 . Similarly, if term2 is a variable and term1 is any type of term, then term1 and term2 unify, and term2 is instantiated to term1 . (So if they are both variables, they’re both instantiated to each other, and we say that they share values.)
3. If term1 and term2 are complex terms, then they unify if and only if:
   1. They have the same functor and arity[[1]](#footnote-1)
   2. All their corresponding arguments unify
   3. The variable instantiations are compatible.

For example, it is not possible to instantiate variable X to mia when unifying one pair of arguments, and to instantiate X to vincent when unifying another pair of arguments.

In other words, if two constants are equal, they unify. Variables unify to anything, and take their value. Functions unify to each other if they have the same number of arguments, and the arguments can be unified to each other.

Prolog unifies two terms by repeated application of the 3 rules above. If no instantiation can be completed, then the unification will fail.

When two variables are instantiated to each other, Prolog creates a dummy variable and assigns them both to the value of that dummy variable. This variable takes the form of 4 unique random numbers preceded by an underscore.

Prolog’s unification is different from most unification algorithms because it skips the ‘occurs’ check. The occurs step checks the instantiation to ensure that the variable being instantiated does not contain itself. In old versions of Prolog, this would cause a memory overflow. However, newer implementations will catch this, and will return a finite representation of an infinite value. So an infinitely recursive query such as X = father(X) will return yes and a finite representation of an infinite set rather than no, which would be returned by a regular algorithm.

Curry is a functional programming language that implements Prolog-style unification. LISP also has some extensions that implement it.

# Datalog: Prolog for Databases

Datalog was designed to allow prolog-style logic programming for databases. As such, its syntax is a subset of Prolog’s. While it has some restrictions relative to Prolog, it is more expressive than SQL. Unlike Prolog, Datalog is truly declarative, as it allows statements to be made in any order. However, it is not a turing complete language, and as such has few applications outside of databases. Extensions exist for Datalog to implement Object-Oriented programming concepts, which allows queries to return objects, making querying in other languages more straightforward.

Some restrictions in Datalog include the following:

* Complex terms may not be used as arguments of predicates.
* Follows the formal logic of stratification. Stratification is essentially the method that computations are ordered. In datalog, a predicate must have a lower than or equal stratification number than it's derivations, unless negation is included, then it must be strictly less than it's derivation:
  + A(x) :- B(x)
  + B(x) :- A(x)
  + Is valid since A(x) can have the same stratification number as B(x) when there is no negation.
  + A(x) :- \+B(x)
  + B(x) :- \+A(x)
  + Is invalid, since A(x) and B(x) must both have lower stratification numbers than each other.
    - \+ is negation in datalog
* Every variable in a statement must be bound, ie. any variable appearing in the head of the statement must also appear in the body.
* Every variable in a statement which appears in a negated literal must also appear in a positive literal.

As a subset of Prolog, Datalog is able to execute queries faster as it is better optimized for databases. It is also guaranteed that datalog queries on finite sets will complete, which makes it more reliable as a query language.

Datalog implements first-order logic, which means that while programs can be very verbose, the logic used to evaluate them is always sound and complete. While it is generally a weaker language than SQL in terms of database creation, Datalog has a few distinct advantages. First and most importantly, datalog implements recursive calls. Additionally, there is no need to make join statements, since the syntax of Datalog allows a predicate to have multiple derivations, allowing implicit joining.

# Backtracking and Non Determinism

OR-non-determinism is a method of computing in which a solution is built up until either a solution is found or a solution becomes impossible. In the event of the latter the program will backtrack until it reaches a point where it can make a different choice to arrive at a different solution. This is the methodology Prolog uses when constructing solutions.

The n-queens problem asks for a distribution of queens on an n by n board in which no two queens are able to take each other while each column of the board has exactly one queen. In other words, it asks for permutations of queens in each column of an n by n board in which no queens are diagonal or horizontal to any other queens.

The following Prolog code will generate solutions for the 8-queens problem using non-deterministic methods.

|  |
| --- |
| solution([]).  solution([X/Y|Others]) :-  solution(Others),  member(Y, [1,2,3,4,5,6,7,8]),  noattack(X/Y, Others).  noattack(\_,[]).  noattack(X/Y,[X1/Y1|Others]) :-  Y =\= Y1,  Y1 - Y =\= X1 - X,  Y1 - Y =\= X - X1,  noattack(X/Y,Others).  member(Item,[Item|Rest]).  member(Item,[First|Rest]) :-  member(Item,Rest).  template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).  ?- template(S), solution(S). |

In comparison, the following Java code solves the n-queens problem using a backtracking technique. Essentially the algorithm follows a possible solution branch in the backtracking tree until it either reaches a solution, or reaches a ‘dead end’. When a ‘dead end’ is reached, the program will backtrack to the last place where it had to choose a branch, and then start down a new branch. It will do this until it finds a path that leads to a solution. When a path leads all the way to a solution, the program prints that solution and then backtracks to where the last branch occurred, and then starts down a new branch. This is very similar to what Prolog does in order to solve the same problem. Prolog chooses a clause that unifies with a goal and then if a clause fails, it uses built in backtracking techniques to return to the place where the last clause was chosen. It will then choose a different clause that unifies with the goal and continue from there.

|  |
| --- |
| class Queen326 {  static int n;  static int[] x = new int[128];  static boolean[] a = new boolean[128];  static boolean[] b = new boolean[128];  static boolean[] c = new boolean[128];  static void PrintSolution() {  for (int col=1; col<=n; ++col) System.out.print( x[col] );  System.out.println();  }  static void gen( int col ) {  for (int row=1; row<=n; ++row)  if (a[row] && b[row+col] && c[row-col+n]) {  x[col] = row;  a[row] = b[row+col] = c[row-col+n] = false;  if (col == n) PrintSolution(); else gen( col+1 );  a[row] = b[row+col] = c[row-col+n] = true;  }  }  public static void main( String[] args ){  n = Integer.parseInt( args[0] );  for (int i=1; i<=2\*n; ++i) a[i] = b[i] = c[i] = true;  gen( 1 );  }  } |

Icon, an ALGOL-like program language supports backtracking similar to Prolog. Certain functions in Icon suspend upon completion, and if resumed because subsequent functions fail, will undo changes made before failing themselves. In this way, they can emulate the backtracking constructs of Prolog. However, Prolog will retain the backtracking tree after finding a result natively, while such functionality would have to be specifically implemented in Icon. In this way, Prolog is much more extensible and allows for much greater functionality out of the box.

As well, some regular expression implementations use non-deterministic backtracking. They do this by testing all expansions of a regular expression in a greedy fashion. The returned result determines on the exact implementation. For example, POSIX implementations will run through all possible expansions to find the longest matching solution, while DOS implementations will run until they find the first result.

# References:

Datalog Stratification: http://stackoverflow.com/questions/12379775/datalog-stratification

History: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7438&rep=rep1&type=pdf>

<http://www.umcs.maine.edu/~cmeadow/courses/cos301/raising-pigs.pdf>

<http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/>

<http://www-03.ibm.com/innovation/us/watson/>

http://www.youtube.com/watch?v=E1pMI6JYkzM&feature=youtube

Non-deterministic & backtracking

Professor Frank Ruskey

<http://msdn.microsoft.com/en-us/library/0yzc2yb0.aspx>

<http://www.javaist.com/blog/2008/11/06/eight-queens-problem-in-prolog/>

Unification

<http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse5>

1. the number of arguments that a function can take [↑](#footnote-ref-1)