November 27, 2013

Logic Programming and Prolog

Prepared For:

Mantis Cheng, CSC 330
University of Victoria

Written By:

Jasmine Mago, Freja Kelleris
Laura Rossetto, Yuma Katayama
Samuel Navarrete, Daniel Oon
Josh Stelting, Max Gunton
Trevor Schiavone, Breanne Gruenke

Table of Contents

1. History of Prolog

1.1 Introduction

The idea for prolog started in the late 60's and early 70's from a collaboration between Alain Colmerauer in Marseille and Robert Kowalski in Edinburgh. Together they discovered that the clausal form of logic could be used to represent formal grammars and that resolution theorem provers could be used for parsing. The following year Kowalski developed the procedural interpretation of Horn clauses. This interpretation later became formalised in the Prolog notation

$H :- B_1, \ldots, B_n.$

Colmerauer, working with colleague Philippe Roussel, used this research as the basis of Prolog, which was implemented in the summer of 1972. The motivation for Prolog was in part to reconcile the use of logic as a declarative representation language with the procedural representation of knowledge. The program logic of Prolog is expressed in terms of relations, represented as facts and rules. Prolog has since branched off into many different dialects, such as Visual Prolog, Edinburgh Prolog, and ISO Prolog.

1.2 Early Applications

Prolog was originally designed to be used in artificial intelligence and computational linguistics applications, such as natural language processing. The first program written in Prolog was a French question-answering system. At its inception Prolog was used heavily in the European Esprit programme, but it didn't start to gain widespread popularity as a language until 1985 when it was used by the Japanese government in the building of the ICOT Fifth Generation Computer Systems Initiative, which was an attempt to create intelligent computers. Today, Prolog's applications are much more far-reaching.

1.3 Recent Applications

Although Prolog is known as being a language used only in academia or for small personalized tasks, there are quite a few examples of applications available on a large

commercial scale. These applications cover a wide range of different domains, including speech, biotechnology, telecommunications and environmental systems. Below are some examples of (fairly recent) products from each of these domains:

Speech

Clarissa is a fully voice-operated procedure browser developed by the NASA Intelligent Systems Division to enable astronauts to be more efficient and to aid in navigating through complex procedures using spoken commands.

Biotechnology

A dispensation order generation algorithm written in Prolog for Pyrosequencing's sequence analysis instruments is used to calculate an instruction sequence based on an input specification. Applications include genetics, drug discovery, microbiology, SNP and mutation analysis, forensic identification using mtDNA, pharmacogenomics, and bacterial and viral typing.

Telecommunication

Ericsson Network Resource Manager allows for configuring and managing complex multi vendor IP Backbone networks. It assists the operator in making decisions when planning, configuring and making configuration changes. The modelling part of the software is implemented in Prolog. The constructed network model, created by analyzing the actual router configurations, is used both for showing a graphical representation and for validating the network.

Environment

The MM4 Weather Modeling System, developed at Penn State University and the National Center for Atmospheric Research, gives short term forecast of meteorological conditions of the continental United States. It is used particularly to model pollutant dispersion.

Similarly, the Air Pollution Control System, developed using the Hungarian product MPROLOG, handles data about the concentration of industrial pollutants in Budapest and other areas of Hungary. The system checks whether the air pollution of working or planned plants is below the permitted level.

1.4 Machines Built for Prolog

Warren Abstract Machine (WAM)

The Warren Abstract Machine was developed by David Warren in 1983. It is specifically tailored to Prolog which consists of an instruction set and a memory architecture. Its principles were based on the DEC-10 Prolog system, which was also created by Warren and his colleagues. The system was built from the Marseille implementation technique of compiling Prolog into a low-level language, and various space-saving measures. The WAM now serves as a target for portable Prolog compilers, and can be implemented adequately in a wide range of hardware. Due to its efficiency, this abstract machine has now become a standard implementation for Prolog.

Transparent Prolog Machine (TPM)

In 1985, Marc Einsenstadt and Mike Brayshaw developed the Transparent Prolog Machine (TPM). The TPM is a Prolog visualization system which displays the structure of a pure Prolog program as an AND/OR tree representation. It visualizes and animates the execution of Prolog programs in slow-motion or in high-speed. Prolog programmers of any level will find this tool very useful, as it will assist them understand the inner workings of the Prolog interpreter or trace a buggy code. When observing a code, it can be viewed at different levels of detail. A selective highlight option enables the user view a buggy code in a context-sensitive way. Additionally, it allows for user-defined abstracted views of the program trace.

---

2. Unification and Logic Variables

Prolog is a logic-based programming language which uses unification extensively. This section will compare pattern matching in functional programming and unification in Prolog, as well as explore other non-logic based programming languages that support unification in a similar way as Prolog.

2.1 Pattern Matching in Functional Programming

Pattern matching is an important tool in the programming world as it allows for clear and highly expressive ways to declare conditional logic. The concept of pattern matching is applied differently in various functional programming languages. However, the idea is usually that pattern matching allows you to look at identifiers in terms of their the values, and compute results based on these values.. Hence pattern matching allows the exploitation of the structure of objects during a function definition.

It can be said that pattern matching is the compiler's ability to compare two expressions in terms of form and content. There are two notable ways of using it:

- One or several variables in a single expression may be assigned to definition for all or part of a data structure..
- A function may have several definitions depending on content and value. In which expressions are used to to initialize the variables that will be used in the overall definition.

As with definition, syntax used in pattern matching will vary from language to language but the following example can be used as a basic guideline for structure.

```
1.  match expr with
2.  | pat1 -> result1
3.  | pat2 -> result2
4.  | pat3 when expr2 -> result3
5.  | _ -> defaultResult
```

In this case, (|) means we are defining a condition, (->) means return the value, if the given condition is true", and the symbol (_) means it matches with everything as it's the default pattern.

A real example of pattern matching in Haskell follows below.

```
1.  fib 0 = 1
2.  fib 1 = 1
3.  fib n | n >= 2
4.  = fib (n-1) + fib (n-2)
```

From this example, there are a few things to note. The list of patterns to match is *ordered*, in that when determining a match, the various functions are called in order. There is no requirement that the clauses be disjoint. In the factorial example above, it should be noted that the second case doesn't specify n>0 as a precondition; the case of n==0 is handled by the first clause. (However, it might be nice if it did, for other reasons - the second pattern will happily match n==-1, which will result in an endless loop as implemented.)
Many examples of pattern matching in Haskell use a match-anything wild-card as the last clause, which matches any and all arguments.

2.2 Unification in Prolog

Unification in Prolog not only unifies two terms, but also completes the essential instantiations to result in the two terms being equal. In this way, unification is an impressive programming tool when coupled with the idea of recursively structured terms.

The unification of two terms is a straightforward procedure based on the following four clauses [1]:

Clauses:

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:

    - They have the same functor and arity, and
    - all their corresponding arguments unify, and
    - 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 .)

4. Two terms unify if and only if it follows from the previous three clauses that they unify.

Examples:	code
1. *In Prolog, an atom of the form 'example' is the same at that of of the form example, so these terms will unify.*	1. ?- 'bob' = bob 2. yes
2. *Unlike Example 1, these terms will not unify. This is because '4' is an atom, whereas 4 is a number. They are not the same.*	1. ?- '4' = 4 2. no

Unification Prolog differs from pattern matching in other languages because it does not use a standard unification algorithm. This is because Prolog does not implement an 'occurs check' step which means that other languages first check to see if a variable occurs in the term, and if it does, the unification is impossible. take for example the following prolog code:

```
1.  ?- mother(X) = X
```

The standard unification algorithm would answer no. Prolog, in its recursive manner, will infinitely expand the sequence to mother(mother(mother(mother...mother(X))))...) and answer yes. In older Prolog implementations, you will get a message indicating a lack of memory to complete the operation.

While at first this may seem unhelpful, unification in Prolog is actually very handy. A programmer would rarely write anything like the above example on purpose, and the omission of the occurs check makes Prolog a lot faster.

As well, Prolog comes with a built-in predicate to perform a standard unification algorithm, unify_with_occurs_check/2, available for use if desired.

2.3 Unification in Non-Logic Programming Languages

It is possible to embed Prolog in many languages, and thus use its unification functionality. Examples of languages with which this is possible are Java, C++, Haskell, and even Python.

The programming language Curry, which can be described as a combination between logic and functional programming, has strong roots in Prolog. As such it includes features which support similar unification methods to Prolog itself.

Another example is the functional language Erlang. While it does not precisely follow Prolog's unification processes, it does pattern match in order to extract data values from composite data such as lists and tuples. Like Prolog, Erlang tries to equal the left hand side of a statement to the right hand side. As a descendant of Prolog, Erlang' unification also has very similar syntax as Prolog.

---

3. Non-Determinism and Backtracking

3.1 Non-Determinism

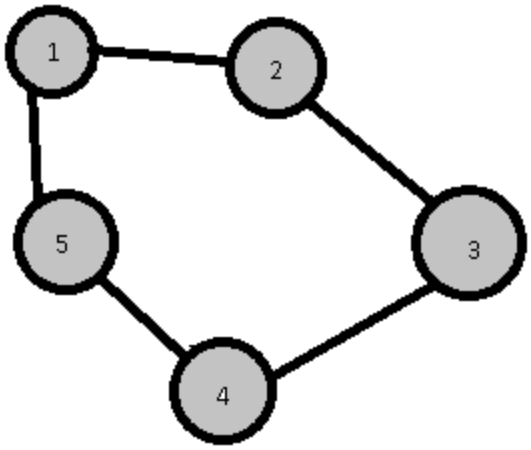
Prolog is non-deterministic. The evaluation of a statement produces multiple computations. A Prolog program will try to solve queries through an evaluation of each goal by comparing rules. Prolog differs from deterministic programming languages, like Pascal, in that at any point there is exactly one next step. In Prolog however, there can be many possible next steps. Upon reaching a point where a goal can't be matched, then backtracking occurs.

3.2 Backtracking

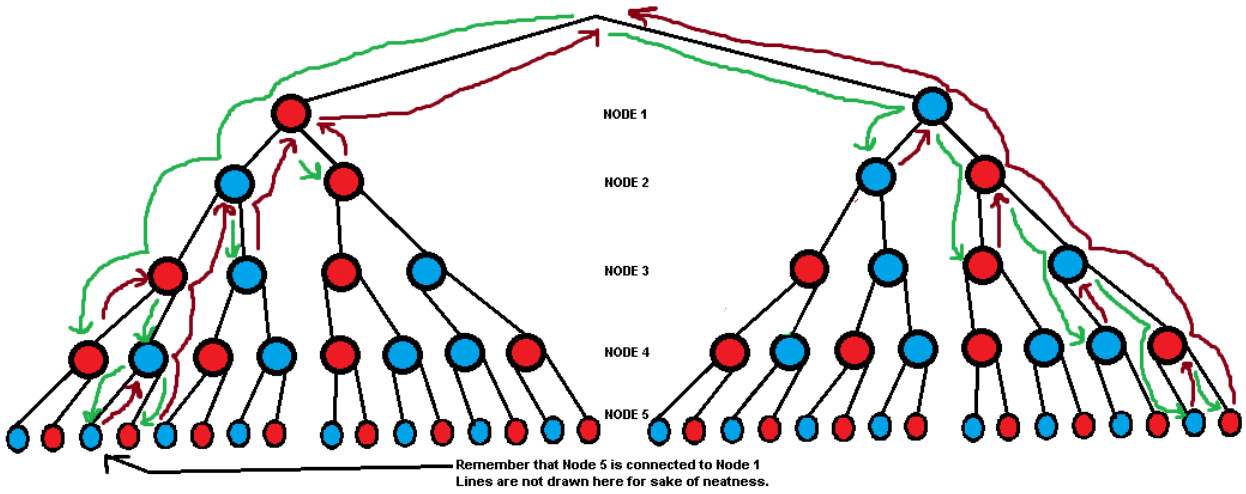
Backtracking is a methodical way of trying out various sequences of decisions until you find one that works. For example, let's say you are traversing a maze. You come to a fork in the maze and can choose to go left or right. Suppose you choose to go left and you eventually come to a dead end. You backtrack to the fork and now go right instead of left. This new path could also lead to a dead end, or provide more choices to make it to the end of the maze.

3.3 Map Coloring Problem

Suppose we are given the map represented with the following adjacency list:
[[1,2] , [2,3] , [3,4] , [4,5] , [5,1]]



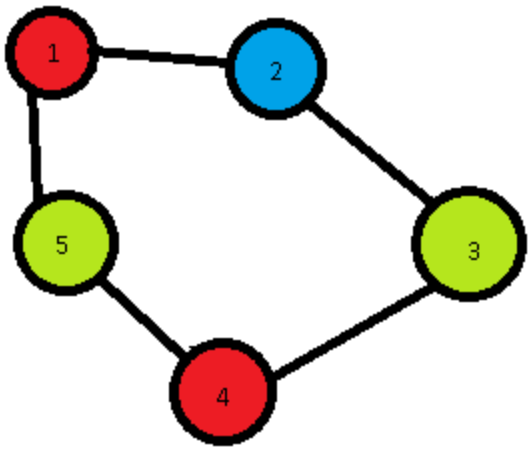
Now, if we were to use non-determination and backtracking, it would look something like the following:



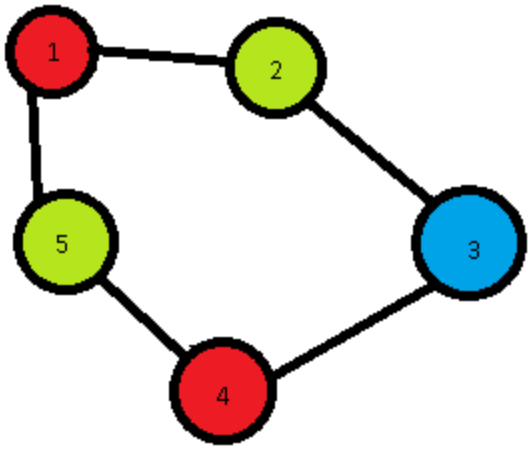
As you can see in the above figure, the algorithm will traverse down the tree as long as it is arriving at a different colour from the previous. If it encounters the same colour as the previous node, it will backtrack and try again from that branch. In this case, the graph is not two-colourable (bipartite).

If we check to see if three colours works, we will find that it indeed does work, and there are multiple possible answers. The tree diagram for the three

coloured graph would too large to fit on this page unfortunately since it has about 243 (3^5) paths. But as you can see, it definitely works:



Example 1



Example 2

3.3.1 Using Prolog

In prolog, this could very easily be programmed with only a couple lines of code.

```
1.   Coloured(A,B,C,D,E) :-  A\=B, B\=C, C\=D, D\=E, E\=A,
2.      colour(A), colour(B), colour(C), colour(D), colour(E).
3.   colour(red).
4.   colour(blue).
5.   colour(green).
```

The above code will take 5 nodes and colour them such that any nodes adjacent to each other are not coloured the same colour. It also restricts it to only three colours.

3.3.2 Using Deterministic Programming Languages

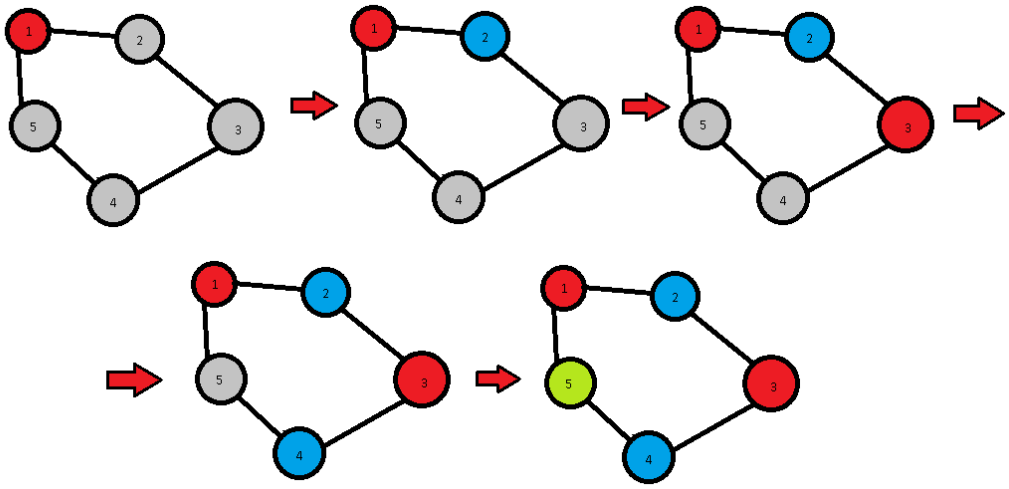
In programming languages that don't support non-determinism problems like the map-colouring one are typically solved via algorithms written into the code by the programmer; rather than at run time by the programming language (via backtracking).

Some of these algorithms for map-colouring include:

Greedy algorithm:

This is guaranteed to work but has the potential to result in the worst case of the number of colours being equal to the number of nodes, even if using a lower number of colours is possible.

The idea is to use the same colour so long as there is no conflict. When conflict occurs the program switches to earliest introduced colour and tries all possibilities from there, and if none work then a new colour is chosen.



Backtracking:

It is possible to implement backtracking in languages that don't support non-determinism, however it has to be written into the code. As seen previously the idea would be to try all possibilities of colouring with 1 colour, then with 2 colours (backtracking until all 2 colouring possibilities are tried), then with 3 colours, etc.

The choice between implementing a greedy algorithm vs using a logical programming language or implementing backtracking is typically made based on the requirements of the user. Greedy algorithms have the potential to be much faster than backtracking since only one pass through the graph is required. The greedy approach to map colouring can be improved on large graphs through algorithms that optimize vertex ordering.

3.4 Other Languages Supporting Non-Determinism

Common Logic Controlled English (CLCE)

CLCE is a formal language, that supports non-determinism, whose syntax closely mirrors English. Anyone with basic english comprehension, and understanding of logical expressions, should be able to understand CLCE. It appears to be the project of John F. Sowa.

An example of the code is:

If some person x is the mother of a person y,
then y is a child of x.

Which has the predicate calculus equivalent of

~(∃x:Person)(∃y:Child)
(Mother(x,y) ∧ ~Child(y,x))

While reading CLCE code is very simple, writing it is much more difficult due to the it's very specific syntactic limitations.

Mercury Programming Language

Developed at the University of Melbourne, Mercury is a "pure logic programming language intended for the creation of large, fast, reliable programs." It's syntax originated as a pure subset of Prolog but has since evolved; the addition of new features has resulted in significantly different syntaxes. Due to features such as modularity, typing, and separate compilation phase Mercury is typically considered a logical programming language more suited towards application development.

---

4. Datalog and Relational Databases

4.1 What is Datalog

Datalog is a declarative logic language used in deductive database systems for queries and updates. It is syntactically a subset of Prolog. Datalog formulas are

function-free Horn Clauses which contain at most one positive literal/fact. A Datalog program is a collection of rules and rules are made of clauses.

4.2 Differences from Prolog and SQL

Although Datalog and prolog share almost all features, there are several differences between them. Unlike Prolog, the Datalog language allows writing the statements in any order. Datalog does not allow to use complex terms as an argument of predicates such that p(f(1), 2). Datalog requires that every variable that appears in the beginning of a clause should also appear in a body of the clause. Though Datalog ensures all queries terminate, it is not guaranteed that an answer is found, unlike the Turing complete Prolog.

Datalog, similarly to Prolog, has two types of queries: one where all arguments are constant and return True or False, and the other which deduces all possible constants that return True to the predicate under the bound variables. Therefore Datalog is syntactically identical to Prolog but not to SQL.

SQL is primarily a server-language paradigm, whereas Prolog is primarily a client-language paradigm. For instance, inferencing would be automatically done in Prolog while in SQL, we have to do it all manually. On the other hand, SQL can deal with storage, retrieval, projection and reduction of trillions of rows with thousands of simultaneous users but Prolog is not designed for those operations.

4.3 Advantages

The main advantage of Datalog is that it allows recursive search operation. For example,

Rules:

```
1.  ancestors(X, Y) :- parent(X, Y)
2.  ancestors(X, Y) :- parent(X,Z), ancestors(Z,Y)
```

Query question:

```
1.  ?- ancestors(X, John)
```

Which basically means “who is/are ancestor(s) of John.” It looks for all of

John's ancestors within the facts/datas. It is unnecessary to have facts called "ancestors(X, Y)" but Datalog can find all the related constants through the rules which defines each predicates.

Datalog is known to be more expressive than SQL. For example, a Datalog clause of "? - father(X, John)" can be represented in SQL using a query along the lines of:

```
1.  SELECT * FROM father WHERE father.child = 'John'
```

Where "father" is the name of a table and "child" corresponds to the name of an attribute in "father." On top of that, an SQL query cannot easily represent a Datalog clause like "? - ancestors(X, John)" unless an extremely large table with all ancestors of John has been entered into the database.

Also, Datalog is able to return all the possible combination of the values of the variables within the clause, given the resulting lists. What it means is,

Rules:

```
1.  append([], Y, Y)
2.  append([u|X], Y, [u|Z]) :- append(X, Y, Z)
```

Query question:

```
1.  ?- append(X, Y, [1,2])
```

Returns:

```
1.  ([], [1,2])
2.  ([1], [2])
3.  ([1,2], [])
```

Given that the resulting list is [1,2], Datalog finds out all of the combination that X and Y can be for the final list. No other languages allow this feature, not even in SQL, which is said to be a similar language.

4.4 The Datalog Educational System

Beyond the benefit of Datalog over Prolog and SQL is the Datalog Educational System, the current standard deductive database language. It has been pure Datalog in the past but it currently includes Datalog, SQL, and Relational Algebra as query languages. Its main advantages are increased expressiveness and clean semantics. Nulls, joins, and more operations are also handled in its Datalog-based languages.

5. References

P. Blackburn, J. Bos and K. Striegnitz. (2006). *Learn Prolog Now!* [Online]. Available: http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse5

E.Czaplicki. *(2013).What is "Pattern Matching"?* [Online]. Available: http://elm-lang.org/learn/Pattern-Matching.elm

D. Sandlers (2011). *Pattern Matching in F#* [Online]. Available: http://www.c-sharpcorner.com/uploadfile/f5b919/pattern-matching-in-fsharp/

C.E. Pramode (2013). *Introduction to Functional Programming with Scala* . [Online]. Available:http://www.slideshare.net/pramode_ce/introduction-to-functional-programming-wih-scala

P. Borek . (2011). A Combination of Functional and Logic Programming [Online]. Available: http://cl-informatik.uibk.ac.at/teaching/ws10/bob/slides/curry.pdf

NA (2013. *Pattern Matching* [Online]. Available:http://c2.com/cgi/wiki?PatternMatching

A. Roth. (2002). *The Practical Application of Prolog*. [Online]. Available: http://www.drdobbs.com/parallel/the-practical-application-of-prolog/184405220

Eisenstadt, M., & Brayshaw, M. (1988). *Program Visualisation and Debugging Logic Programs: The Development of the Transparent Prolog Machine (TPM)* [Online]. Available: http://projects.kmi.open.ac.uk/kmi-misc/tpm/tpm.html

Fernando Saenz-Perez. (2011). *Outer Joins in a Deductive Database System*. [Online]. Available: http://lbd.udc.es/jornadas2011/actas/PROLE/PROLE/S5/13_article.pdf

Hassan Ait-Kaci. (1999). *Warren's Abstract Machine:A Tutorial Reconstruction*. [Online] Available: http://wambook.sourceforge.net/wambook.pdf

Jay McCarthy. (2013). *Datalog: Deductive Database Programming*. [Online]. Available: http://docs.racket-lang.org/datalog/

Grigoris Karvoumarakis. (2013). *DATALOG*. [Online]. Available: http://users.ics.forth.gr/~gregkar/publications/encyclopedia-datalog.pdf

SICS AB. (2013) *SICStus Prolog Customer References*. [Online]. Available: http://sicstus.sics.se/customers.html

SOURCEFORGE.NET. (2013). *Datalog Educational System*. [Online]. Available: http://www.fdi.ucm.es/profesor/fernan/des/

Mercury Project (2013). *Mercury Programming Language*. [Online]. Available: http://mercurylang.org/

Mercury Project (2013). *About Mercury*. [Online]. Available: http://mercurylang.org/about.html

Wikipedia. (2013). *Logic Programming*. [Online]. Available: http://en.wikipedia.org/wiki/Logic_programming

Wikipedia. *Prolog*. [Online] Available: http://en.wikipedia.org/wiki/Prolog

Wikipedia contributors. (2013). *Datalog*. [Online]. Available:http://en.wikipedia.org/wiki/Datalog

Wikipedia. (2013). *Mercury (programming language)*. [Online]. Available: http://en.wikipedia.org/wiki/Mercury_(programming_language)