Take-home Midterm 3: Prolog
Group B

Q1: History of Prolog

The name "Prolog" is derived from PROgramming and LOGic and was created at the University of Aix-Marseille in 1972 by Alain Colmerauer. Robert A. Kowalski's research (Edinburgh University) set the theoretical framework for Prolog which helped Colmerauer create the language. Philippe Roussel chose the name as an abbreviation for Prolog to refer to the software tool designed to implement a man-machine communication system in natural language. It can be said that Prolog was the offspring of a successful marriage between natural language processing and automated theorem-proving. The idea of using a natural language like French to reason and communicate directly with a computer seemed like a crazy idea, yet this was the basis of the project set up by Alain Colmerauer in the summer of '70. Alain had some experience in the computer processing of natural languages and wanted to expand his research.

Invention steps

1971: the first step

At the beginning of 1971, everyone was back in Marseilles. Alain had obtained a teaching position in Computer Science (maître de conférence) and Jean Trudel had been able to accompany him, thanks to a two-year grant from Hydro-Quebec. The aim of the project was to make deductions based on texts written in French. The work was divided up as follows: Jean Trudel and Philippe worked on the deduction part, Robert Pasero and Alain on the natural language part.

Jean Trudel improved his prover and then, beginning in May, Philippe produced a whole series of provers written in Algol-W. A primitive natural-language communication system was then developed by the entire team [Colmerauer, 71]. The interfaces between the logical formulæ and French consisted of 50 Q-system rules for the input and 17 Q-system rules for the output. The reasoning part was implemented through one of Philippe's provers. It was thus possible to have the following conversation with the computer:

> **User:** *Cats kill mice. Tom is a cat who does not like mice who eat cheese. Jerry is a mouse who eats cheese. Max is not a mouse. What does Tom do?*
> **Computer:** *Tom does not like mice who eat cheese. Tom kills mice.*
> **User:** *Who is a cat?*
> **Computer:** *Tom.*
> **User:** *What does Jerry eat?*
> **Computer:** *Cheese.*
> **User:** *Who does not like mice who eat cheese?*
> **Computer:** *Tom.*
> **User:** *What does Tom eat?*

Computer: *What cats who do not like mice who eat cheese eat.*

1972: The application that created Prolog

During the fall of 1972, the first Prolog system was implemented by Philippe in Niklaus Wirt's language Algol-W; in parallel, Alain and Robert Pasero created the eagerly awaited man-machine communication system in French [Colmerauer, 1972a]. There was constant interaction between Philippe, who was implementing Prolog, and Alain and Robert Pasero, who programmed in a language which was being created step by step. This preliminary version of Prolog is described in detail in Part III of this paper. It was also at this time that the language received its definitive name following a suggestion from Philippe's wife based on keywords which had been given to her. The man-machine communication system was the first large Prolog program ever to be written [Colmerauer, 1972]. It had 610 clauses: Alain wrote 334 of them, mainly the analysis part; Robert Pasero 162, the purely deductive part, and Henry Kanoui wrote a French morphology in 104 clauses, which makes possible the link between the singular and plural of all common nouns and all verbs, even irregular ones, in the third person singular present tense.

Here is an example of a text submitted to the man-machine communication system in 1972:

> ***Every psychiatrist is a person. Every person he analyzes is sick. Jacques is a psychiatrist in Marseille. Is Jacques a person? Where is Jacques? Is Jacques sick?***

and here are the answers obtained for the three questions at the end:

> ***Yes. In Marseille. I don't know.***

The original text followed by the three answers was in fact as follows:

> *TOUT PSYCHIATRE EST UNE PERSONNE.*
> *CHAQUE PERSONNE QU'IL ANALYSE, EST MALADE.*
> **JACQUES EST UN PSYCHIATRE A *MARSEILLE.*
> *EST-CE QUE *JACQUES EST UNE PERSONNE?*
> *OU EST *JACQUES?*
> *EST-CE QUE *JACQUES EST MALADE?*
> *OUI. A MARSEILLE. JE NE SAIS PAS.*

1973: the Final Prolog

Between February and April 1973, at the invitation of Robert Kowalski, Philippe visited the School of Artificial Intelligence at the University of Edinburgh, which was within the Department of Computational Logic directed by Bernard Meltzer. Besides the many discussions with the latter and with David Warren, Philippe also met Roger Boyer and Jay Moore. They had constructed an implementation of resolution using an extremely ingenious method based on a structure-sharing technique to represent the logical formulæ generated during a deduction. The result of this visit and the laboratory's need to acquire a true

programming language prompted our decision to lay the foundations for a second Prolog.

Early Applications of Prolog
Colmerauer and Phillipe Roussel built the first Prolog interpreter, and David Warren at the University of Edinburgh built the first Prolog compiler (for the DEC-10 machine). Most commercial implementations of Prolog now use this same Edinburgh Prolog syntax.

Early on Prolog was used at York in the Student Information System to check applications for input errors and was widely used to develop expert systems & other AI applications including natural language processing.

Other early applications

- PigE- A system to help raise pigs which won the best presentation award at the first international prolog conference, it was able to run simulations of pig growth to help maximize profits.
- The Air Pollution Control System, developed using the Hungarian product MPROLOG, handles data about the concentration of industrial pollutants in Budapest and other countries of Hungary. Designed for such users as managers and research workers, the system checks whether the air pollution of working or planned plants is below the permitted level. If permitted levels are exceeded, the system recommends appropriate filtering equipment.
- GoldFinder - Knowledge-based system that advises geologists on where to find gold, made with MacProlog.

More on PigE(AusPig)
PigE led to some startling returns on investment. PigE discovered that increasing the airflow over a hot pig shed by 0.2-0.6 meter per second (an increase not really discernable by humans), doubled the animals' growth rate. In one spectacularly successful study of a pig farm, Auspig was used to develop a growing regime that led to 496% increase in profits (from $40,992 to $202,231).

According to the developers of the system, verification studies have demonstrated that the expert system can significantly outperform human experts interpreting the output of the model (performance measured in dollars per square meter per day). In an unusual case, the improvement is in the order of 10%. If spread over the 300 kiloton $500 million (per annum) Australian pig herds, the developers believe that this improvement would be in the order of $50 million. PigE was in routine use in the 90s in the United States, the Netherlands, Belgium, France, Spain, and Australia.

Machines and Applications of Prolog recently
Some applications of Prolog are natural language understanding, expert systems(ie. Distributed Systems), specification language,machine learning, robot planning, automated reasoning, problem solving. A more recent application of logic programming can be found in

artificial intelligence. The machine designed is a Supercomputer.

On February 14-16, 2011, the IBM Watson question answering system won the Jeopardy Man vs. Machine Challenge by defeating two former grand champions, Ken Jennings and Brad Rutter. To compete successfully at Jeopardy!, Watson had to answer complex natural language questions over an extremely broad domain of knowledge. Moreover, it had to compute an accurate confidence in its answers and to complete its processing in a very short amount of time. Engineers used Prolog to express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness. The information in the parse is easily converted into Prolog facts, such as (the numbers representing unique identifiers for parse nodes):

```
lemma(1, "he").
partOfSpeech(1, pronoun).
lemma(2, "publish").
partOfSpeech(2, verb).
lemma(3, "Songs of a Sourdough").
partOfSpeech(3, noun).
subject(2, 1).
object(2, 3).
```

Such facts were consulted into a Prolog system and several rule sets were executed to detect the focus of the question, the lexical answer type and several relations between the elements of the parse. A simplified rule for detecting the authorOf relation can be written in Prolog as follows:

```
authorOf(Author,Composition) :-
        createVerb(Verb),
        subject(Verb,Author),
        author(Author),
        object(Verb,Composition),
        composition(Composition).
createVerb(Verb) :-
        partOfSpeech(Verb,verb),
        lemma(Verb,VerbLemma),member(VerbLemma, ["write", "publish",...]).
```

Another application of prolog is Clarrissa - A voice user interface made by NASA for the International Space Station used for browsing space station procedures 2005. Clarissa has been implemented mainly using SICStus Prolog and a speech recognition toolkit provided by Nuance Communications. Application-specific spoken command grammars were constructed using the SICStus Prolog based Regulus platform.

Q2: Unification and Logic variables

Pattern Matching
In Haskell, pattern matching is used to bind values to different parts of a defined variable.

ex: *sum [] = 0*
sum (x:xs) = x + sum xs

Now, calling the sum function could either result in 0, or x + sum xs. This depends on the length of the list that is passed during the call of sum, an example of pattern matching.

Unification in logic programming is the ability to compare two terms, complex or otherwise, and determine if they can be evaluated to the same truth value.

ex: *father(X) = father(john) evaluates to yes when X is instantiated to john*

Unification
In Prolog, terms can be either constants or variables. Two terms will unify if and only if they meet the following criteria:

- If two terms are constant, they will unify only if they are the same atom, or have the same number.
- A variable and a constant will always unify.
- If the two terms are complex terms, they will unify if and only if:
  - They have the same functor and arity
  - All their corresponding arguments unify
  - The variable instantiations are compatible

The arity of a term is the amount of arguments that it takes as input.

ex: *In the statement: f(X, Y) = f(X), f(X, Y) has arity 2, and f(X) has arity 1. These two terms cannot unify for any value of X.*

Prolog not only tells you if two terms can unify, it will also tell you which constants will allow the terms to unify.

ex:
```
fish(red).
fish(blue).
| ?- fish(X).
X = red
X = blue
yes
```

This can be very powerful when used with complex terms to determine which clauses can complete an equation.

Differences between Unification and Pattern Matching
The difference between pattern-matching and unification is that pattern-matching is a one way operation while unification is two-way. This means, with unification, unbound variables can be put on either side of a comparison. In a functional programming language, the statement X = Y will cause an error because Y is an unbound variable. In Prolog, this statement will unify X and Y. It should be noted that for X and Y to unify, they must be instantiated to the same term, and that Prolog will output the possible terms for which this will occur.

Pattern matching also depends on the order of its arguments. Functional languages type check along the way, and the argument types must be in the same order as the definition of the function. Prolog does not type check in this way so switching the order of variables will not result in an error.

ex: Haskell: *take 0 _ = []*
take _ [] = []
take n (x:xs) = x : take (n-1) xs

take1 _ [] = []
take1 0 _ = []
take1 n (x:xs) = x : take (n-1) xs

Using take and take1 with the call 'take 0 list' or 'take1 0 list'. take returns the empty list, while take1 is undefined.

Prolog: *f(X,Y) = f(john,phil) if X = john, and Y = phil*
f(Y,X) = f(john,phil) if X = phil, and Y = john
Both of these will evaluate to yes but with different values for X and Y.

Unification in Non-Logical Programming Languages
Type inference in many functional languages, such as Haskell and ML, bears similarities to unification in Prolog.

- Any type variable unifies with any type expression, and is instantiated to that expression. A specific theory might restrict this rule with an occurs check.
- Two type constants unify only if they are the same type.
- Two type constructions unify only if they are applications of the same type constructor and all of their component types recursively unify.

OCaml is a multi-paradigm language that takes influence from functional, and imperative languages. Its method of pattern matching is more similar to Prolog unification in that it is a two-way comparison of two expressions. The unifications are done using specific

substitutions similar to facts in Prolog.

ex: *'f x (g y)', 'f (g z) w'*. These two expressions unify after a substitution:
S = [x <- g z, w <- g y]

Most functional languages don't support unification itself. Unification algorithms must be implemented manually for these languages.

Q3: Non-determinism and backtracking

Languages which support non-determinism are able to evaluate multiple "choices" to see which can provide suitable solutions to a problem. During execution, if the program makes a choice which 'fails', it can return to the choice point and make another choice. This is known as backtracking.

Magic Squares
A magic square is an arrangement of integers in an n x n square where the numbers in each row, column, and diagonal add up to the same number. No numbers may repeat so it will always contain the numbers 1 through n^2. All examples in this report will use a 3x3, or order 3, magic square. Below shows an empty magic square of order 3 (left) and a solution for it (right).

X_1	X_4	X_7
X_2	X_5	X_8
X_3	X_6	X_9

8	1	6
3	5	7
4	9	2

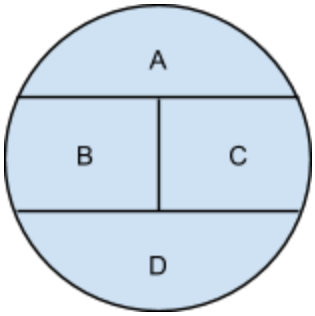
Using Prolog, the magic square problem can be implemented by listing a set of rules. These rules include having no digit be equal, and having all rows add to the same number. Also given is which numbers are in a row, column, or diagonal line together. Because Prolog uses nondeterminism, the program chooses the numbers from a list given by the programmer, as opposed to having conditions and comparisons hard coded by the programmer. Prolog is also capable of cycling through multiple solutions to this problem. An example of Prolog code for this magic square problem is as follows:

```
% Find all 3 by 3 magic squares.
% d(D) is for digit
d(1). d(2). d(3). d(4). d(5). d(6). d(7). d(8). d(9).
% a row has 3 different digits that add up to 15
row(X,Y,Z):-d(X),d(Y), X=\=Y, d(Z),X=\=Z, Y=\=Z, X+Y+Z=:=15.

col(X,Y,Z):-X=\=Y, Y=\=Z, X=\=Z, X+Y+Z=:=15.

diag(X,Y,Z):-X=\=Y, Y=\=Z, X=\=Z, X+Y+Z=:=15.

printrow(A,B,C):-write(A),write(B),write(C),nl.

out(A,B,C,D,E,F,G,H,I):-nl,
    printrow(A,B,C),
    printrow(D,E,F),
    printrow(G,H,I).

go:-    row(A,B,C),
        row(D,E,F),
        row(G,H,I),
        col(A,D,G),col(B,E,H),col(C,F,I),
        diag(A,E,I), diag(G,E,C),
        out(A,B,C,D,E,F,G,H,I).
```

Programming languages that do not have nondeterminism will require comparisons and conditions hard coded by the programmer as opposed to letting the program choose through nondeterminism. While the prolog solution to the problem is only 23 lines in length, a Haskell solution is about 60 lines long (ex. http://pastebin.com/0xPaxWzh). A java solution that's 35 lines long uses a pattern that is explained within the comments (ex. http://introcs.cs.princeton.edu/java/14array/MagicSquare.java.html). To write a solution to the magic square problem in a language without nondeterminism requires longer code and/or implementing a pattern like mentioned within the Java example. The pattern is somewhat of a short cut to a solution. With or without the pattern discovered, many comparisons and conditions must be written by the programmer, often leading to longer code. Most programming languages without nondeterminism will only produce a single solution, unless implemented to do otherwise, while prolog will cycle through multiple solutions given time. A problem with rules, limitations, and multiple possible solutions such as the magic square problem is preferable in prolog, as it is easily written and efficient.

Map Coloring

The Map-Coloring problem refers to coloring a map such that no adjacent or connected regions are given the same color. For example:



In order for this to be a valid map coloring solution, three conditions must be met:

1. region A differs from region B and C
2. region B differs from region C
3. region D differs from region B and region C as well.

Of course, in more complicated problems, the number of conditions would grow substantially - the more regions, and the more boundaries, the more rules you need to create.

In PROLOG, this is a much easier problem than in other languages like Python or C or Java. Because of backtracking, when one of the conditions of a valid solution fails, the program only has to step back once or twice and then proceed with another "guess." In order to generate a different answer, PROLOG programs backtrace to a point at which they can make another decision - one that differs from a decision made previously - and steps forward completing the program.

```
1   Color(Red)
2   Color(Blue)
3   Color(Green)
4   Color(Yellow)
5
6   Map(A, B, C, D) :- Color(A),
7                      Color(B),
8                      Color(C),
9                      Color(D),
10                     A != B, A != C, B != C, D != B, D != C.
```

A more general solution would involve parsing an set of pairs of letter representing borders. An example "function definition" would be
Map([[1,2],[1,3],[1,4],[1,5],[2,3],[2,4],[3,4],[4,5]]) representing 5 regions with boundaries between 1 and 2, 1 and 3, etc.

A comparable program written in Python would look like the following:

```python
colors = ['Red', 'Blue', 'Green', 'Yellow']

regions = ['A', 'B', 'C', 'D']

neighbors = {}
neighbors['A'] = ['B', 'C']
neighbors['B'] = ['A', 'C', 'D']
neighbors['C'] = ['A', 'B', 'D']
neighbors['D'] = ['B', 'C', 'E']

final_colors = {}

def valid_coloring(region, color):
    for neighbor in neighbors.get(region):
        color_of_neighbor = final_colors.get(neighbor)
        if color_of_neighbor == color:
            return False

    return True

def get_color_for_region(region):
    for color in colors:
        if valid_coloring(region, color):
            return color

def main():
    for region in regions:
        final_colors[region] = get_color_for_region(region)

    print final_colors

main()
```

One of the issues with this, and one of the benefits of a language like PROLOG, is that, due to the non-determinism, you can keep polling the program for another answer. Subsequent executions of this Python program will always return the same answer as the program halts as soon as it finds a valid solution:

```
w87-164-237:Code Samples tills13$ python map_coloring.py
{'A': 'Red', 'C': 'Green', 'B': 'Blue', 'D': 'Red'}
w87-164-237:Code Samples tills13$ python map_coloring.py
{'A': 'Red', 'C': 'Green', 'B': 'Blue', 'D': 'Red'}
w87-164-237:Code Samples tills13$ python map_coloring.py
{'A': 'Red', 'C': 'Green', 'B': 'Blue', 'D': 'Red'}
w87-164-237:Code Samples tills13$ python map_coloring.py
{'A': 'Red', 'C': 'Green', 'B': 'Blue', 'D': 'Red'}
w87-164-237:Code Samples tills13$ python map_coloring.py
{'A': 'Red', 'C': 'Green', 'B': 'Blue', 'D': 'Red'}
```

A program could be written in Python to accomplish this, you would simply have to iterate through the various possibilities for a given region, which would cascade possible colorings for other regions.

```
def main():
    for i in range(len(regions)):
        final_colors[regions[0]] = colors[i];
        for j in range(len(regions) - 1):
            final_colors[regions[j + 1]] = get_color_for_region(regions[j + 1])

        print final_colors

main()
```

While this solution only generates possible answers for different colors for region A, it demonstrates that there's no good way to poll the program for one solution, and if you want another solution, you have to modify your code. It's also difficult to obtain one solution at a time. If you want to read many solutions, the easiest option is to compute all of them and then display them.

SEND + MORE = MONEY

The "SEND + MORE = MONEY" problem involves solving the equation, obviously, SEND + MORE = MONEY. Numeric values of 0 - 9 are assigned to variables S, E, N, D, M, O, R, and Y such that the equation is true.

```
Digit(0)
  ...
Digit(9)

[S,E,N,D,M,O,R,Y]
Digit(S)
  ...
Digit(Y)

M > 0,
S > 0,
1000*S + 100*E + 10*N + D +
1000*M + 100*O + 10*R + E =:=
10000*M + 1000*O + 100*N + 10*E + Y.
```

In another language, this could be accomplished thusly:

```
def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            if eval(equation):
                return equation
```

while this is a more complicated example which takes **any** alphametic puzzle and solves it, it can be applied to this specific problem. The highlighted section is the "meat" of the solution. In this solution, permutations of all the possible solutions are generated, and then checked to see if a valid solution has been generated. In order to generate another possible solution, the algorithm would have to be run again, with this previous solution remembered and checked against before returning the potentially valid solution. PROLOG programs can generate one, and using the values from the first solution as a basis, attempt to generate another solution. This can be repeated, more or less, *ad infinitum* (until all solutions have been generated).

Icon

Icon, (by Ralph Griswold) is a language based on a non-deterministic 'pass' or 'fail' control structure instead of a 'true' or 'false' boolean based structure. Expressions which 'fail' do not produce output. This allows decisions to be evaluated using a choice based approach. For example: *while write(read())* will continue to read an input file and output it to a single line. When the *read()* command fails (when it reaches the end of the file) the failure will be passed on to the write command which will also will also fail (it will no longer be able to write). This allows the entire operation to complete using only three words of code. Indeed, Icon is known for it's compact, efficient code.

Screamer

Screamer (by Jeffrey Mark Siskind and David Allen McAllester) is a Lisp extension which adds support for nondeterministic programming. It adds the operator 'either' which creates a choice point which is immediately followed by a list of options. For example, *either 1 2 3 4* creates 4 choices, one for each integer 1, 2, 3, and 4. An *if* statement can test each choice for failure. If the test fails, the 'fail' operator will cause the system to backtrack to the previous choice point. While Screamer is able to undo some of the side effects cause by backtracking, they are not undone by default. Calls to functions cannot be undone.

Q4: Datalog and Relation Databases

Datalog owes a great debt to Prolog, and the logic programming area in general. It is a logic programming language based on predicate calculus, which is a form of symbolic logic (which is the method of representing logical expressions through the use of symbols and variables, rather than in ordinary language, for example: ¬ (negation), ∧ (and), ⊕ (XOR), etc).

Datalog arose from **Horn Clause Logic** with two restrictions:

- Function symbols are disallowed: terms must be variables or be drawn from a fixed set of constant symbols.

- Any variable in the head of a clause also appears in the body.

The Horn Clause Logic is basically a clause (disjunction of literals), where at most one of the literals is positive, and all the others are negative. There are three types of clauses:

- **Definite Clause**: A Horn clause with exactly one positive literal.
  For example: ¬p ∨ ¬q ∨ ... ∨ ¬t ∨ u
- **Goal Clause**: A Horn clause without a positive literal.
  For example: ¬*p* ∨ ¬*q* ∨ ... ∨ ¬*t*
- **Fact**: A definite clause with no negative literals. For example, u, (read simply as "assume that u holds")

The Horn Clause plays a basic role in constructive logic and computational logic and was used to implement the programming language **Prolog.**

From Prolog, came Datalog, which is a query and rule language for deductive databases, which is simply a database system that can make deductions (i.e., conclude additional facts) based on rules and facts stored in the (deductive) database.

Datalog exists somewhere between relational algebra, which is a formal system for manipulating relations (and is also the formal theory behind **SQL**), and Prolog.

Datalog was invented to apply some of the principles of logic programming to database theory. Its primary addition to the semantics of databases is **recursive** queries. This makes Datalog more **expressive** than SQL, which does not support recursive queries *(you can do it but it's a lot harder and you need to used stored procedures, which is a subroutine available to applications that access a relational database system, and it's stored in the database data dictionary.)*

Is it declarative?
Yes! Datalog is a truly declarative logic programming language. Datalog focuses more on *what* the program should accomplish, in terms of the problem domain, rather than describing *how* to go about accomplishing it. For example, in datalog, a query specifies what the result should be (using logic) rather than how to compute it (using algebraic operators). Procedural (imperative) languages, such as Java or Pascal or C, describe solution steps instead.

Some differences between Datalog and SQL:

- Datalog rules and queries can be recursive, unlike SQL. This makes datalog much more expressive, readable. and easier to implement solutions for, than SQL.
- Datalog adds two more main advantages over SQL:
  First, its clean semantics allows to better reason about problem specifications. Its neater formulations, especially when using recursive predicates allow better understanding and program maintenance.

Second, Datalog provides more expressivity because the linear recursion limitation from SQL is not imposed.

Some differences between Prolog and Datalog:

- Prolog is Turing Complete, which means it can do anything that any Turing machine can do, if it had all the time, memory, and software that it needs. In other words, turing complete means 'able to answer any computable problem given enough time and space'. However, Datalog (and SQL) is not Turing Complete
  An advantage of having a language that is non-Turing complete such as Datalog, could for instance be that it might be sufficient to perform the tasks you need, while being simple enough to allow you to prove properties about your programs, which you could not otherwise prove. This could, for instance, be useful in cases where it's vital to know that the program will run without error (and terminates).

- Prolog allows complex terms as arguments of predicates. For example, f(1,2) is allowed in both but p(f(1),2) is allowed in Prolog, and not in Datalog. The advantage to Prolog having this feature is that you can parametrize functions over things that aren't easily expressed as plain values, and it allows you to write cleaner, more general code.

- Datalog limits assignments that are possible under recursion and negation (*stratification*). Stratification is a consistent assignment of numbers to predicate symbols guaranteeing that a unique formal interpretation of a logical theory exists. Prolog does not have this stratification restriction.

- Datalog only allows range-restricted variable. That is, each variable in the conclusion of a rule must also appear in a non-negated clause in the premise of the rule. This is a safety feature in Datalog which is not implemented in Prolog and gives an advantage to Datalog over Prolog.

These rules also have the consequence that all datalog programs will terminate, unlike Prolog programs. Also, as a consequence, statements and predicates of a program can be stated in any order unlike in Prolog.
In conclusion, Datalog provides benefits over Prolog and SQL in ways such as implementing safety conditions of only allowing range-restricted variables, being able to provide recursive queries, and being a very readable and expressive programming language.

Sources

http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse5
http://en.wikipedia.org/wiki/Horn_clause
http://rosettacode.org/wiki/Category:Datalog
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7438&rep=rep1&type=pdf
http://www.drdobbs.com/parallel/the-practical-application-of-prolog/184405220
http://en.wikipedia.org/wiki/Declarative_programming
http://www.nasaspaceflight.com/2005/06/iss-set-to-get-chatty-with-clarissa/
http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/
http://stackoverflow.com/questions/2117651/difference-between-sql-and-prolog
http://nikodemus.github.io/screamer/#Overview
http://www.csci.csusb.edu/dick/cs620/lab19.html
http://www.cs.rochester.edu/u/kautz/Courses/444au2010/prolog-assign.html
http://www.cs.arizona.edu/icon/index.htm
http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec26-type-inference/type-inference.htm#3
http://lbd.udc.es/jornadas2011/actas/PROLE/PROLE/S5/13_article.pdf