University of Victoria

Department of Computer Science

Logic Programming and Prolog

Prepared by the A Team for CSC 330

Josh Braendel
Jessica Paquette
John Rogers
Marvin Messing
Ryan McDonald
Gabriel Letourneau
Catlin Zilinski
Jessica Blue
Hiroki Yun
Bill Xiong

Table of Contents

1. History of Prolog

1.1 Who invented it and how? What are its early applications?

Prolog is a general purpose, goal-oriented logic programming language currently associated with artificial intelligence and computational linguistics [8]. Prolog was born out of a project aimed at processing natural languages such as French rather than producing a programming language. It was Philippe Roussel who chose the name as an abbreviation for "**PRO**grammation en **LOG**ique," meant to reference their implementation of a man-machine communication system in natural language [6]. There are currently many applications for Prolog along with several variations on the original language, such as SWI-Prolog and GNU-Prolog, which are both Free Software implementations.

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.

Fig. 1: First conversation with computer based on progress of research:

“Prolog is a high-level programming language which enables the user to build programs by stating **what** they want the program to do rather than **how** it should do it.”
-- The University of Edinburgh School of Informatics course page

Prolog emerged from a collaboration between Alain Colmerauer in Marseille and Robert Kowalski in Edinburgh [1] beginning in 1970. The first program written in Prolog (a simple question-answer system) was deployed in France during 1972. Colmerauer and Kowalski’s Edinburgh version became the de-facto standard to which all Prolog versions adhere to, including GNU-Prolog, which is still in use today. Colmerauer and Phillipe Roussel built the first Prolog interpreter, and David Warren at the University of Edinburgh was credited with building the first Prolog compiler (for the DEC-10 machine) [9].

The first large implementation of a Prolog program was the man-machine communication system. It had 610 total clauses; there were 104 written as a French morphology, making the singular and plural of all common nouns and verbs possible in the third person singular present tense. In 1973, laboratory users of the preliminary version of Prolog created a second version more firmly oriented as a programming language and “not just a kind of automated deductive system” [6].

Prolog was one of the first logic programming languages, and remains the most popular of this type of language today. Its inventor, Alain Colmerauer, was recognized in 1997 by the Association of Logic Programming as a pioneer in this area of computer science [2]. While initially aimed at natural language processing, Prolog has bridged into fields such as theorem proving, expert systems, games, automated answering systems, ontologies, and sophisticated system controls [4]. Modern Prolog environments even support the creation of graphical user interfaces, as well as administrative and networked applications. A course on Prolog and its applications in Artificial Intelligence is still taught at the University of Edinburgh.

The initial version of Erlang was implemented in Prolog [10]. Since being open sourced by Joe Armstrong, Erlang has been used to great success by many of the major software and digital services companies, among them Amazon to implement SimpleDB, Yahoo and its social bookmarking site del.icio.us, and Facebook chat. Erlang is prized in the webapp industry for its ability to handle multiple connections simply and effectively, its asynchronous nature, and its ability to “scale horizontally” across multiple machines as your project increases in size and scope. Although Erlang has separated into its own language, the influence of Prolog is still very evident in its syntax.

1.2 Recent Applications

1.2.1 IBM Watson

On February 14-16 2011, the IBM Watson Question-Answering system won the Jeopardy Man vs. Machine challenge by defeating Brad Rutter and Ken Jennings, two former grand champions. Watson used Natural Language Processing algorithms to parse a Jeopardy question into trees. For example, given the question "he was a bank clerk in the Yukon before he published 'Songs of a Sourdough' in 1907," Watson was able to determine that "published" is a verb with "publish" as its base form, "he" is the subject, and "Songs of a Sourdough" is the object. The machine then used numerous detection rules to match patterns in the parse. Watson was built using Prolog due to its ability as a language to conveniently express pattern matching rules over parse trees and other annotations, as well as execute these rules very efficiently [7]. Watson won very handedly in both games against Mr. Rutter and Mr. Jennings, and is now being used commercially in the workplace, healthcare and finance. Specifically, Watson is employed making utilization management decisions in lung cancer treatment at Memorial Sloan–Kettering Cancer Center [11].

Figure 2. Watson with his human underlings.

"Cognitive systems like Watson may transform how organizations think, act, and operate in the future. Learning through interactions, they deliver evidence based responses driving bett outcomes." --IBM [12]

1.2.2 PigE/Auspig

Another interesting recent application of Prolog is PigE/Auspig—an expert system for raising pigs. Auspig is a mathematical modelling package which "simulates the growth and reproduction of pigs, identifies factors that limit optimal performance of the pig, and suggests management strategies that maximize profits" [9]. PigE is an intelligent extremity which displays the output simulations from Auspig in a non mathematical way while suggesting dietary, housing, genotyping, or resource input changes[9] based on the data. Verification studies done on the expert system by developers indicate a significant increase in accuracy of data interpretation over human experts.This has led to drastic returns on investment and greatly increased the profitability of the herd. 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). PigE is now regularly used in the United States, the Netherlands, Belgium, France, Spain, and Australia.

1.2.3 Environment

Environmental systems are also a thriving application of Prolog, from predicting weather conditions to analyzing water supply. One example is RoadWeather, a 24-hour weather prediction system used for snow and ice control on highways. Developed using PDC Prolog version 3.30 for Windows 3.1, RoadWeather Pro consists of three components: a numerical weather prediction system written in FORTAN, a graphical user interface (GUI) written in Prolog under Windows 3.1, and an Expert Weather Advisor, written in Prolog and Windows 3.1, which permits forecast upgrades based on recent observational data received from sensors or human observers. The RoadWeather Pro system is in use at the WELS Research Corp. in Boulder, Colorado.

1.2.4 Sports

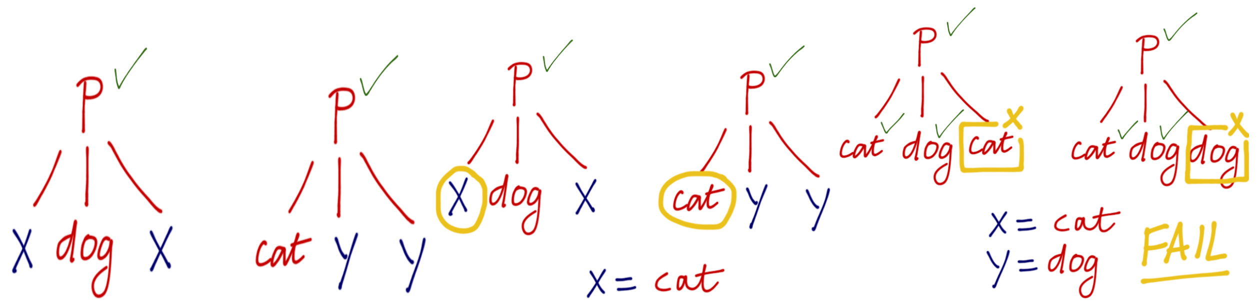
Prolog can be used for more recreational applications as well as commercial. John Dowding of Palo Alto California created a Rotisserie/Fantasy Basketball league system that automatically retrieves and stores information in a Prolog database, which ranked the players and teams. Dowding says “Prolog has been an excellent vehicle for this task because of its built-in support for databases, sorting, and parsing” [9].

J.G. de Lint from the Netherlands used Prolog to create Toernooi Assistant, a tennis tournament automation system. It provides planning and replanning of matches, player administration, seeding and drawing of lots for the arrangement of the players, processing of the match results, generation of press releases, and financial support [9].

1.2.5 Goldfinder

On a smaller scale, the Prolog application Goldfinder was created by D.D. Hawkes from the London-based Logic Programming Associates for the purpose of mineral prospecting. Goldfinder uses a map input by the user (typically an exploration geologist) to assess the best drilling location [9].

2. Unification



A Simple Illustration of the Unification Algorithm

2.1 Unification in Prolog vs Pattern Matching in Functional Programming

Unification in Prolog and pattern matching in functional programming have one key difference between them. The main thing that sets pattern matching apart is that it is unidirectional. This feature of pattern matching makes it a popular choice for certain problems including string and list searching, regular expressions, and structural pattern matching. For instance, structural pattern matching involves finding matches between predicates; e.g. male(Dave), male(X), {X <- Dave}. An example of the limitation in pattern matching is if we test one pattern, male(X), against another pattern, male(Y), then Y must be an atom term (a.k.a. a terminal, or a constant term, e.g. male(Dave)). On the other hand, unification is an algorithm for solving pattern matching in a bidirectional manner. This means that expressions can occur in both patterns during the matching test. Essentially we are given two patterns and look for matches between each of the terms to check if the two patterns match. For example, the pattern append(A, [2], [1,2]) could be matched with append([1], B, [1,2]) to find that {A <- [1]}. Prolog's unification algorithm works by taking two patterns, forming a list of pairs of each of the terms, and then checking that each pair is a match. We can check if each pair of terms (p,q) is a match with the following case-by-case check routine:

1. if p and q are both atoms (ie. terminals, or constants), then p == q implies that the pair is a match
2. if p is a variable (eg. p = X), and q is an atom, then assign q to p, and restructure both patterns by replacing all instances of p with q
3. if q is a variable (eg. q = X) then apply a symmetrical equivalent of case 2
4. if p and q are both variables, (eg. p = A, q = B) then replace all instances of p with q

If all pairs of terms match between the two patterns, then the two patterns are considered to match. Prolog's unification algorithm can be found in [13]. Unification is a very powerful feature, if not the most powerful feature of Prolog and other logic based programming languages; however, unification isn't necessary in all pattern matching implementations. Functional programming languages often adopt a variety of pattern matching forms: structural pattern matching, string matching and regular expressions. However, the limitation of functional programming pattern matching is that it is unidirectional.

2.2 Other non-logic based programming that also supports Unification

There are indeed other languages which are non-logic based, that use unification. These other languages don't rely on it to the extent that Prolog does, as Prolog is completely based on it, but they do use it for things such as type inference. Haskell is a prime example of this. Haskell uses unification as a mechanism to infer an object type (if it is obvious). Haskell uses an algorithm that follows an extremely similar structure as Prolog [14]:

1. Instead of variables which are uninstantiated being unified with a term or another uninstantiated variable, Haskell unifies type variables with any type expressions. Thus the type variable becomes instantiated to that expression.
2. Instead of two constants unifying if they're identical we have two type constants unifying only if they are the same type.
3. Instead of terms with the same arities being unified, in Haskell two type constructions unify only if they are applications of the same type constructor and all of their component types recursively unify.

There are some less known functional languages that use unification. One example of these is HASL or HArvey's Static Language. HASL is a purely functional language that uses unification for conditional binding expressions. HASL is a direct descendent of SASL (St. Andrew's Static Language) and was created simply to add unification based expression binding. HASL uses the exact same algorithm as PROLOG for unification and it is implemented through it's function cMATCH. For more information on HASL please see the research paper in [15]

An example of a functional language that uses unification more as an extension than as an integrated part is Common Lisp. Common Lisp is a dialect of the original Lisp and is a multi-paradigm programming language. Common Lisp includes a library with a 'full blown' [16] framework for unification. Common Lisp implements unification as a process that makes sure that two object description containing some variable can be made equal by assigning consistent values to the variables involved. The process's resemblance to the algorithm used in Prolog is evident. The unification library is accessed through the function named **unify.**

3. Non-determinism and Backtracking

3.1 Introduction

A program is considered non-deterministic if, at any point in execution, there is more than one possible step that may be taken. Programming languages such as C are deterministic, as a program written in C has exactly one possible output for a set of possible inputs. Prolog, however, is nondeterministic. During a Prolog program's execution, there may be multiple valid paths that the program may take to produce a valid output. [17]

To illustrate the strength of nondeterminism, we will explore Prolog's uses in natural language processing. In particular, we will discuss the implementation of several simple parsers using Prolog, and compare the methods used to those of more common programming languages.

First, let us review the definition of a parser. The goal of a parser is to analyze the syntax of a program. Parsers must take as input a sequence of symbols, find the syntactical structure of these symbols, and determine whether the given program is syntactically correct or not.

The syntactic structure of a given program can be represented as a *context-free grammar* (CFG); that is, a grammar consisting of

- terminal symbols
- nonterminal symbols
- productions
- a start symbol

[18]

Modern parsers are implemented either as *top-down*, or *bottom-up*. Both of these methods generate a *syntax tree* for the given input. We define a top-down parser as a parser who generates a syntax tree beginning with the start symbol of the CFG, and a bottom-up parser as a parser which generates a syntax tree beginning with the input word of the program being analyzed. [19]

3.2 Creating a Parser in Prolog

Prolog provides several very elegant methods of parser generation. In this section we implement a simple parser using the "append" method.

Let us consider a BNF grammar for some simple language:

```
<vardecl>    ::=  var <ident> <typespec>
<ident>       ::=  <letter> { <idchar> }
<idchar>     ::= <letter>  |  <digit>
<typespec> ::= int | string | float
```

Fig 1. A simple BNF grammar

Consider a *vardecl* to be a list comprised of "var", an *ident* and a *typespec.* If this pattern is followed for each part of the grammar, then we can very easily represent this grammar in Prolog by using append.

```
is_var('var').
typespec('int').
typespec('string').
typespec('float').

vardecl(Z)        :-  is_var(P), ident(Q), typespec(R), append(P, Q, S), append(S, R, Z).

ident(Z)          :- is_char(X), idchar(Y), append(X, Y, Z).
                   :- is_char(Z).

idchar(Z)         :- is_char(Z).
                   :- is_digit(Z).
```

Fig 2. Code for a simple parser in Prolog using append

The above Prolog code is an example of a top-down parser known as a *recursive descent* parser. Input is scanned from left to right, verifying syntax as terminals and nonterminals appear in the code. The ease of representation lies in Prolog's method of execution. *Definite Clause Grammars,* covered later, provide an even simpler method of parser implementation.

A Prolog program naturally creates a tree, where all paths from the root to any given leaf in the tree represents a valid answer to a question. The above parser represents a depth first search of the tree. That is, for some rule there is more than one possible path to a leaf, but such paths are only explored if the first rule fails. If a rule fails at any point, Prolog will backtrack to the so-called "junction", and try the next available rule. [20]

3.3 Parsers in Other Languages

3.3.1 Haskell

The Parsec library for Haskell allows users to generate various parsers by combining higher-order Combinators into more complex expressions. [21]

For example, the following EBNF grammar

```
expr   ::= expr '+' term | term
term   ::= term '*' factor | factor
factor ::= '(' expr ')' | digit+

digit  ::= '0' | '1' | ... | '9'
```
Fig 1. An EBNF grammar for expressions [22]

may be expressed using Parsec as

```
import ParsecExpr

expr    :: Parser Integer
expr    = buildExpressionParser table factor
        <?> "expression"

table   = [[op "*" (*) AssocLeft, op "/" div AssocLeft]
          ,[op "+" (+) AssocLeft, op "-" (-) AssocLeft]
          ]
        where
          op s f assoc
             = Infix (do{ string s; return f}) assoc

factor  = do{ char '('
            ; x <- expr
            ; char ')'
            ; return x
            }
        <|> number
        <?> "simple expression"

number  :: Parser Integer
number  = do{ ds <- many1 digit
            ; return (read ds)
            }
        <?> "number"
```
Fig 2. Parsec-style recursive descent parser in Haskell [22]

Like the parser written in Prolog, the Parsec-style parser maintains modularity and faithfulness to the provided grammar. However, the technique used is very different from that of the Prolog-written parser.

Parsec uses a powerful method called *combinator parsing.* Combinator parsing considers a "parser" as a function from a string of values to a result value. Parsers return a list of pairs as their result, where the empty list denotes failure and a singleton denotes success. Several special parsers are denoted as primitives, representing the empty string, success, failure, symbol recognition, and so on. These parser functions are then combined together by combinators, in order to create the desired parser for the given grammar. The goal of combinator parsing is to generate higher-order functions which correspond exactly to the rules provided by the EBNF grammar. [23]

3.3.2 C

Parsers in C are typically implemented using a compiler compiler, such as YACC or GNU's Bison. Hand-written parsers in C can be error-prone and difficult to debug, but are used when human input provides better optimization than that provided by the generator. GCC previously used a parser automatically generated by Bison, but now makes use of a hand-written compiler. [24]

C-written parsers, unlike the previously shown parsers in Prolog and Haskell, are not written in such a way that preserves the style of the EBNF grammar.

3.4 Definite Clause Grammars

Definite clause grammars (DCG's) represent a grammar as a set of definite clauses in first-order logic.[25] A definite clause grammar has one term on the left-hand side, and 0 or more terms on the right-hand side. An example of a definite clause grammar for the language over $a^n cb^n$ is given below.

```
S→[c].
S→[a], S, [b].
```

A definite clause grammar recognizer for $a^n cb^n$***.***

In the definite clause grammar defined by S, the terminals (a, b and c) are represented in the square brackets.

Definite clause grammar rules and Prolog clauses are very similar, they both have one term on the left-hand side, and zero or more on the right-hand side.[26] Because of this similarity, Prolog is very well suited for the task of parsing with definite clause grammars. Definite clause grammar rules can be easily translated into Prolog clauses. This translation results in the creation of a recursive descent parser, which uses difference lists.[26] A difference list is a way of representing a list by the difference between two lists. In Prolog, difference lists are represented with the argument pair ([L | X], X).[25] The Prolog translation of the DCG for $a^n cb^n$ is shown in the following example..

```
S([c|L0], L0).
S([a|L0], L1) :- S(L0, [b|L1]).
```

Prolog clauses representing a DCG recognizer for $a^n cb^n$.

The Prolog clauses, and their corresponding definite clause grammar rules, represent a recognizer, i.e., a program that is able to recognize if the input is valid according to the grammar rules. In translating the definite clause grammar to prolog clauses, → is replaced by :-. The two parameters for each predicate represent the input being parsed as difference lists. Each clause consumes as much input as it can recognize from the front of the input and returns the rest of the input to be parsed as the output parameter. In general, the translation from a definite clause grammar rule to prolog clauses is as follows[28]:

Definite clause grammar rule:
alpha(args) → beta_1(args) … beta_n(args)

Prolog clause:

```
alpha(Arg_1,...,Arg_m, P0, Pn) :- beta_1(arg_i1,1, …, arg_i1,k1, P0, P1),
                                   ….,
                                   beta_n(arg_in,1, …, arg_in,kn, Pn-1, Pn).

% Where alpha is a non-terminal, and the beta’s are either terminal or non-terminal
```

Translation from DCG rule to Prolog clause

Even though definite clause grammars are based on context-free grammars, DCG’s can also be used to represent context-*sensitive* grammars by adding additional parameters to the definite clause grammar rules.[26] The typical context-sensitive grammar, $a^n b^n c^n$, is shown below as a definite clause grammar.

```
S(N) → A(N), B(N), C(N).
A(0) → []
B(0) → []
C(0) → []
A(succ(N)) → [a]A(N).
B(succ(N)) → [b]B(N).
C(succ(N)) → [c]C(N).
```

Definite clause grammer recognizer for the context sensitive grammar $a^n c^n b^n$.

The additional parameter, N, added to the DCG rules works to count the number of a’s b’s and c’s, the main grammar rule, S, ensures that the numbers are equal, giving us the context-sensitive grammar defined by $a^n b^n c^n$.

Definite clause grammars can be used not only to construct recognizers, but also to parse and generate lists.[27] Prolog’s phase/2 predicate can be used in conjunction with grammar rules to generate lists corresponding to valid symbols in the grammar. For example, if we take our definite clause grammar rule for $a^n cb^n$, we can get Prolog to generate all the possible symbols accepted by the grammar with the following command:

```
?- phrase(S, L).
yes.
L = [c].
L = [a, c, b].
….
```

Example of generating valid symbols for the DCG for $a^n cb^n$.

The first argument is the body of a definite clause grammar, and the predicate is only true if the elements of the list L make up a valid symbol in the grammar defined by S. Leaving L as a variable will make Prolog output all the possible lists L that contain a valid symbol in the grammar. Explicitly specifying what L is will cause Prolog to return yes if the grammar S describes L, or no if L is not in the grammar.

```
?- phrase(S, [a, a, c, b, b].
yes.

?- phrase(S, [a, a, c, b]).
no.

?- phrase(S, [a, a, X, b, b]).
yes.
X = c.
```

Example of testing for valid symbols with the DCG for $a^n cb^n$.

Generating parse trees is also quite easy with definite clause grammars in Prolog. To do this, all we need to do is add an argument to the DCG rules, which will keep track of the structure of the parse tree while the input is being parsed.[25] Suppose we have the following DCG representing a very simple sentence.

```
s      → np, vp.
np     → det, noun.
det    → [the].
noun → [person].
vp     → verb.
verb  → [programs].
```

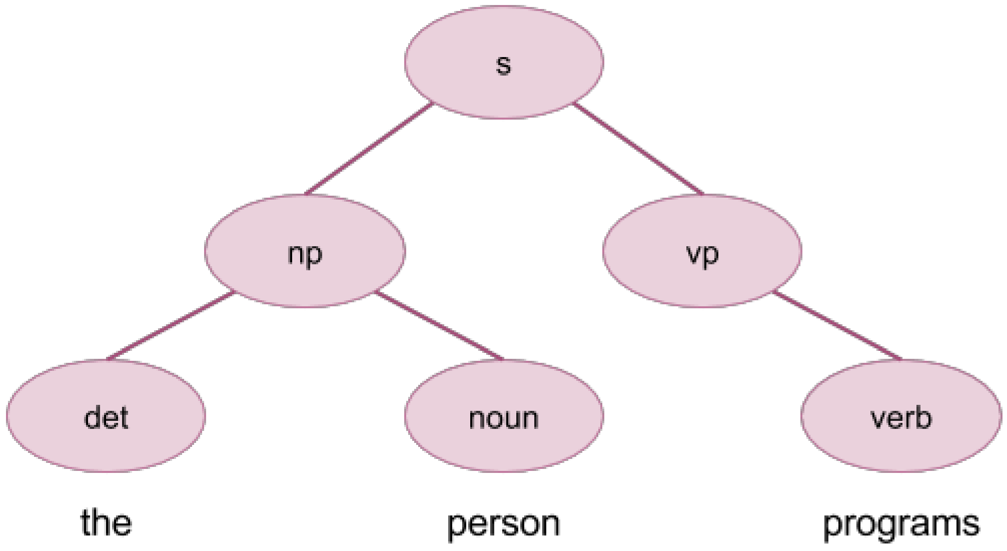
Definite clause grammar for very simple english sentence.

This DCG will match the sentence “the person programs”. Not very useful in the general sense, but good enough for our purposes. To be able to generate the parse tree, we add an argument to each of the rules, which will keep track of the structure of the tree as the input is being parsed.

```
sentence(s(NP, VP))            → nounphrase(NP), verbphrase(VP).
nounphrase(np(DET, NOUN))  → det(DET), noun(NOUN).
det(det(the))                     → [the].
noun(noun(person))            → [person].
verbphrase(vp(VERB))          → verb(VERB).
verb(verb(programs))          → [programs].

?- sentence(T, [the, person, programs], []).
yes.
s(np(det(the), noun(person)), vp(verb(programs))).
```
Definite clause grammar parser for very simple english sentence.

The parse tree Prolog returns is equivalent to the following tree:



3.5 Definite Clause Grammars in Other Languages

Prolog's built in backtracking and non-determinism make parsing DCG's extremely easy, providing the ability to quickly and easily write parsers for both context-free and context-sensitive grammars with Prolog. Creating the equivalent parsers in other languages such as Haskell or Python is much more difficult. Building recognizers for specific grammars can be done; however, building parsers for DCG's makes heavy use of unification and backtracking, which makes it extremely difficult in non-logic programming languages. The most common method to solve this is actually to embed Prolog in another language, giving that language the ability to parse DCG's. This was done in the book *Paradigms of artificial intelligence programming*, by Peter Norvig, where he embedded Prolog into LISP for Prolog's ability to parse definite clause grammars.

4. Datalog and Relational Databases

In 1977 David Maier, a professor at Portland State University, coined the term ‘Datalog’ [29] that describes a syntactic subset of Prolog. Unlike Prolog queries, any Datalog query will have a finite number of answers; because of this, Datalog queries can resolve more quickly than Prolog queries - especially recursive queries [30]. This speed makes Datalog especially suited to write high level logic programs for querying graphs or relational structures and for formal reasoning and analysis [30]. Initially, there was much interest in Datalog, but it was seen to have little practical use [31]. Today, Datalog is extended for commercial interests and used in the fields of data integration, data extraction, program analysis, security, and cloud-computing [30]. This type of logical programming is also being marketed to domain experts and business people as means of automating decision-making [32].

4.1 How Datalog Differs from Prolog and SQL

The syntaxes of Datalog and SQL differ greatly. Databases in SQL are made up of tables and tuples, which can be queried by using built-in keywords. Comparatively, Datalog databases are made up of facts, and can be queried by declaring rules and then asking the database to fill in partially completed rules. The example below shows two queries, SQL on the left and on the right the Datalog dialect IRIS, that both have the result ‘masi’. In terms of power the two are comparable because both are based on predicate logic and what could be achieved in one could theoretically be achieved in the other [33]. Traditionally, SQL would be used to handle large amounts of data that would be subject to create / read / update / delete commands from many sources [35], while Datalog would be used by academics on a small scale [34]. Today, Datalog is being used in fields that traditionally use SQL, like storing and intelligently querying large datasets to discover potential efficiencies that a corporation may capitalize on [32].

```
CREATE TABLE bikes
(name varchar(255), frame_size int);
INSERT INTO bikes VALUES (‘jamis’,
56);
INSERT INTO bikes VALUES (‘masi’, 55);
SELECT name FROM bikes WHERE
frame_size=56;
```

```
bike('masi', '55').
bike('jamis', '56').
getBike(?x, ?y) :- bike(?x, ?y).

?-getBike(?x, '56').
```

There are several differences between Prolog and Datalog. For example, the basic data-log language does not support negation (the difference of two sets). This severely limits the number of queries that can be answered using Datalog. However, there are versions of Datalog that support negation [35]. Datalog also does not allow function symbols as arguments of predicates.

Example: father(john, f(joe)) is not possible in Datalog.

Prolog uses goal-orientated top down approach, Datalog uses bottom-up approach. Prolog programs can be optimized by modifying the order of the prolog statements (rules or facts). This, however, cannot be done in databases where a query may request any combination of rules (previously unknown to the programmer), therefore, Datalog is independent of the database content. Fortunately, Datalogs bottom-up approach can be optimized without changing the order of the database content. Even though the bottom-up approach can result in redundant computations, with optimizations such as the magic set transformation it will be at least as fast as the top down approach. Prolog can run into infinite loops, while datalog cannot, this makes datalog purely declarative, since no procedural elements (such as Prologs “cut” operator) are needed to change the control flow [36].

4.2 The Advantage of Datalog over Prolog and SQL

Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog. [39] The main restriction that distinguishes Datalog from Prolog is that function symbols are not allowed, meaning only constants and relational symbols are allowed. This is considered less expressive than Prolog. Instead, Datalog takes its advantage over Prolog by negated atoms in the body of the rules called stratified semantics that impose syntactic restrictions on the use of negation and recursion, which is natural and relatively easy to understand. [40]

Datalog allows users to express queries by describing the desired results, rather than by giving an algorithm to compute the results, as relational algebra requires in SQL.[42] All queries that can be expressed in core relational algebra can also be expressed in Datalog. Note that core relational algebra does not include some functional transforms and a variety of aggregation operations (group by, count, min, max, top N, etc.) [38] The most powerful feature is recursive Datalog that can express queries that are not possible to write in SQL. [41]

Works Cited

[1] Robert A. Kowalski. 1988. The early years of logic programming. Commun. ACM 31, 1 (January 1988), 38-43. DOI=10.1145/35043.35046
http://doi.acm.org.ezproxy.library.uvic.ca/10.1145/35043.35046
[2] "Association for Logic Programming." Association for Logic Programming. N.p., n.d. Web. 27 Nov. 2013.
http://www.cs.nmsu.edu/ALP/the-association-for-logic-programming/alp-awards/
[3] "Artificial Intelligence Programming in Prolog (AIPP)." Artificial Intelligence Programming in Prolog (AIPP). N.p., n.d. Web. 27 Nov. 2013.
http://www.inf.ed.ac.uk/teaching/courses/aipp/
[4] Stickel, M.E. A Prolog technology theorem prover: implementation by an extended Prolog compiler. Journal of Automated Reasoning 4, 4 (December 1988), 353-380.
http://download.springer.com/static/pdf/139/art%253A10.1007%252FBF00297245.pdf?auth66=1385776041_11784d5bd166b5e12b72a7e079295101&ext=.pdf
[5] "SWI-Prolog's Home." SWI-Prolog's Home. N.p., n.d. Web. 27 Nov. 2013.
http://www.swi-prolog.org/
[6] Alain Colmerauer and Philippe Roussel. 1992. The birth of Prolog.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7438&rep=rep1&type=pdf
[7] "Association for Logic Programming." Association for Logic Programming. N.p., n.d. Web. 27 Nov. 2013.
http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/
[8] "Prolog." *Wikipedia*. Wikimedia Foundation, 27 Nov. 2013. Web. 27 Nov. 2013.
http://en.wikipedia.org/wiki/Prolog
[9] "The Practical Application of Prolog." *Dr. Dobb's*. N.p., n.d. Web. 27 Nov. 2013.
http://www.drdobbs.com/parallel/the-practical-application-of-prolog/184405220
[10] "Erlang." *Wikipedia*. Wikimedia Foundation, 27 Nov. 2013. Web. 27 Nov. 2013.
http://en.wikipedia.org/wiki/Erlang_(programming_language)
[11] Upbin, Bruce. "IBM's Watson Gets Its First Piece Of Business In Healthcare."*Forbes*. Forbes Magazine, 08 Feb. 2013. Web. 27 Nov. 2013.
http://www.forbes.com/sites/bruceupbin/2013/02/08/ibms-watson-gets-its-first-piece-of-business-in-healthcare/
[12] IBM Corporation. IBM Watson. 2013. Web. 27 Nov. 2013
http://www-03.ibm.com/innovation/us/watson/
[13] Michael Spivey, *An Introduction to Logic Programming through Prolog*, Prentice-Hall International, 1995.
http://spivey.oriel.ox.ac.uk/wiki/files/lp/logic.pdf
[14] “Unification.” *Wikipedia*. Wikimedia Foundation, 2002
http://en.wikipedia.org/wiki/Unification_%28computer_science%29

[15] Harvey Abramson, *A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions*, University of British Columbia, 1983
http://download.springer.com/static/pdf/420/art%253A10.1007%252FBF03037050.pdf?auth66=1385782843_e2f8be7b3b532598e92a9f2998d4f07b&ext=.pdf
[16] Drew Crampsie, *Common Lisp Extensions: UNIFICATION*,
http://common-lisp.net/project/cl-unification/
[17] D. Warren, *Nondeterminism,* Stony Brook University, 1999
http://www.cs.sunysb.edu/~warren/xsbbook/node6.html
[18] *Context-Free Grammars,* Rochester
http://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html
[19] D. Maurer and R. Wilhelm, *Compiler Design*. Addison Wesley 1995
[20] Patrick Blackburn, *Top Down Parsing*
http://cs.union.edu/~striegnk/courses/nlp-with-prolog/html/node46.html#l6.sec.topdownparsing
[21] “Parsec.” *Parsec Homepage*, 2005
http://legacy.cs.uu.nl/daan/parsec.html
[22] “Parsec, a fast combinator parser.” *Parsec Homepage,* 2005
http://legacy.cs.uu.nl/daan/download/parsec/parsec.html
[23] Graham Hutton, *Higher-Order Functions for Parsing* , University of Utrecht
http://eprints.nottingham.ac.uk/221/1/parsing.pdf
[24] “GNU Bison.” *Wikipedia*
http://en.wikipedia.org/wiki/GNU_bison
[25] "Definite clause grammar." *Wikipedia*. Wikimedia Foundation, 27 Nov. 2013. Web. 27 Nov. 2013.
http://en.wikipedia.org/wiki/Definite_clause_grammar
[26]Jacques Cohen, Timothy J. Hickey. 1987. *Parsing and Compiling Using Prolog*. DOI=10.1.1.101.9739
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.9739&rep=rep1&type=pdf
[27] *DCG’s in Prolog*, Marakus Triska, 27 Nov. 2013. Web. 27 Nov 2013.
http://www.logic.at/prolog/dcg.html
[28] *Natural Language Analysis,* Horsch, 27 Nov. 2013. Web. 27 Nov 2013.
http://www.cse.buffalo.edu/faculty/alphonce/.OldPages/CPSC312/CPSC312/Lecture/LectureHTML/CS312_10.html#6
[29] - Abiteboul, Serge; Hull, Richard; Vianu, Victor, *Foundations of databases*, p. 305
[30] - Huang, "Datalog and Emerging applications" (PDF),*Sigmod*, UC Davis.
[31] - Ceri, S; Gottlob, G; Tanca, L (1989), "What you always wanted to know about Datalog (and never dared to ask)", *Transactions on Knowledge and Data Engineering* (IEEE) **1** (1): 146–66.
[32] LogicBlox. http://www.logicblox.com. Accessed Nov 2013
[33] - StackOverFlow. http://stackoverflow.com/questions/2117651. Accessed Nov 2013
[34] - Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P. The CORAL deductive system. The VLDB Journal 3, 2 (1994), 161{210.
[35] - Springer. http://link.springer.com/chapter/10.1007%2F3-540-58907-4_8#page-1. Accessed

Nov 2013
[35] - MySQL. http://www.mysql.com. Accessed Nov 2013
[36] - Datalog Bottom Up. http://ttnz.cz/statnice/04%20Datalog/other_docs/hinz.pdf. Accessed Nov 2013
[37] - http://bluehawk.monmouth.edu/~rscherl/Classes/KF/ull.pdf
[38] - Query Language Comparison. http://c2.com/cgi/wiki?QueryLanguageComparison
[39] - Rosseta Code. http://rosettacode.org/wiki/Category:Datalog
[40] - Grigoris Karvounarakis, *DATALOG* p.1-2
http://users.ics.forth.gr/~gregkar/publications/encyclopedia-datalog.pdf
[41] - *Recursive Programming in Datalog*
http://infolab.stanford.edu/~ullman/dscb/dl-old.pdf
[42] - Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom *Database Systems The Complete Book*