Midterm 3

Team F
November 27th, 2013

Jeremy Mohr
Robert Aftias
Lee Gauthier
Tim Ritenour
Mustafa Abualsaud
MacKay McGillivray
Siduo Guan
Mohammed Alghamdi
Francis Harrison

History

Development of Prolog and Early Applications

Prolog was created in Marseille, France in the early 1970s. Its creation is mainly credited to Alain Colmerauer, however there were a number of people involved. Prolog was given its name by Philippe Roussel as an abbreviation of “Programmation en Logique” (Programming in Logic). The name refers to the implementation of using a natural language to communicate with a machine. Alain came up with the idea of using a natural language (French) to interact with a computer in 1970. He had past experience with the idea and wanted to expand his research. Q-systems, a system developed by Alain, played a large role in the creation of Prolog. During early 1970 Alain Colmerauer, Robert Pasero, and Philippe Roussel researched the computer processing of natural languages at the University of Montreal, before moving to Marseille. A former student of Alain, John Trudel, joined the group in their move from Canada back to France where they would further their work on the project. In 1971 the group worked on an IBM 360-44 located at the Marseille University computing centre. The machine had 900Kb of internal memory and had an operating system without virtual memory.

For a week in June 1971, the group was visited by Robert Kowalski. Kowalski was a specialist in automated theorem-proving and discussed his research with the group. The interface between French and logical formulae used 50 Q-system rules for the input and 17 for the output. This made it 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.
```

In 1972 the group obtained a grant of 122,000FF abling them to purchase a teletype terminal and connect it to an IBM 360-67. During this time Kowalski visited the group again for an extended period of time. Philippe Roussel implemented the first Prolog system in the language Algol-W. This was in tandem with Colmerauer and Pasero's man-machine communication system in French. This system was the first large Prolog program to be written; it contained 610 clauses. Here is an excerpt from an early Prolog program which calculated plane schedules:

```
+FLIGHT(<departure city>,<arrival city>,
<departure time>,<arrival time>,<flight identifier>)
```

Flight times are represented by a pair of integers in the format <hours>:<minutes>. Users could call upon the following predicate to plan a route:

```
PLAN(<departure city>,<arrival city>,<departure time>,<arrival time>,
<departure min time>,<arrival max time>)
```

In 1973 the French National Centre for Scientific Research (CNRS) officially recognized the group as an associated research team, entitled “Man-Machine dialogue in natural language”. The CNRS provided funding which furthered the development process. During this time the language’s syntax, basic primitives, and the interpreter’s computation methods were further developed. Postgraduate students Gérard Battani, Henry Méloni, and René Bazzoli wrote the interpreter in Fortran and its supervisor in Prolog. Prolog was further developed by minimizing concepts and improving its interactive capabilities in program management. The implementation of a set of built-in predicates made it possible to do everything in the language itself.[1]

Alain Colmerauer initial intentions were not to create a programming language, but rather a means of processing natural language. As a result, there weren’t many practical applications during its early days. The main purpose of Prolog was to allow man-machine communication using a natural language. Prolog was primarily used in research applications. Not many commercial applications were found until long after its development. Some of the more practical applications were for database retrieval and problem solving.[2] Early implementations of prolog enabled its users to perform a number of tasks: the creation and modification of programs in memory; the ability to read, analyse and load source programs in memory; interpreting queries dynamically; dynamic access to the elements and structure of a deduction; and simple control over program execution.[1] More modern versions of prolog are used in various aspects of artificial intelligence.

Recent Applications

Comparing with imperative programming, one of the main advantages of modern logic programming is that most errors and bugs are caught during compile time, instead of run time. Any programmer who has experience in imperative programming would know how hopeless and annoying it is to catch bugs

at runtime. However, traditional logic programming, represented by Prolog, has its serious drawbacks, such as cut predicate and false negations; fortunately, those drawbacks are virtually eliminated in modern logic programming languages.[5] This gives modern logic programming wider application areas, such as theorem proving, expert systems, games, automated answering systems, complicated scheduling situations, warehousing problems, ontologies and sophisticated control systems. [3][4]

One of the modern logic programming applications is expert systems. Expert system are a part of artificial intelligence, which emulates the decision-making ability of a human expert. [6] Basically, it's just a program telling AI "if the outside world do THIS to you, you respond with THAT action." Naturally, this is derived from logic programming's original purpose: human-computer interaction. It is correct to assume that the so-called "expert system" is just a set of if-else code. For a novice programmer, he might be picturing an expert system in C-style like this in his mind:

```
if (input == "hello")
{
    printf("hey man you are awesome!\n");
}
else if (input == "you look ugly, you know that?")
{
    printf ("Yeah you look good man, but how come you're still single?\n");
}
```

However, people rarely write expert system using imperative languages' conditional statements because of 2 reasons. First, expert systems are all about logic; this means when designing AI's interaction with outside world, the conditions, responses, and overall logic will quickly become overwhelmingly complicated, thus too hard or even impossible to implement in imperative languages (compared with Prolog, C's conditional statements are like caveman's work). The second reason to use logic programming language is that it allows expert system to establish new facts using rules, and use new facts and rules to find solutions. This means that AI can think and learn at some level. It is very hard, and likely impossible for imperative language to implement.

However, that does not mean that logic languages, modern or traditional, are absolutely perfect for expert system. One feature that spoils things is closed-world-assumption (CWA). Imagine I ask a guy called Tom, "Hey Tom, does your friend Jerry go to the same high school with you?" Assume that Tom doesn't know where his friend Jerry went to high school, and Tom does not know all the people in his high school, he would answer "I don't know." That is how people think, and it's called open-world-assumption (OWA), meaning we admit that the part of the world we know might not be the whole world, and we don't make a decision based on things we don't know; most imperative languages do the same thing as well. However if you ask an expert system developed under CWA the same question, under the same assumption about not having everyone's name in high school in his database, the expert system will answer "No". It is because for CWA, he assumes that the world he knows is the whole world, and his list of high school students must be the whole list (although this can be false, but he doesn't care); if his database doesn't contain an entry called "Jerry", then Jerry must not have gone to his high school.

The effects of CWA for expert systems are one of the main drawbacks for logic programming. The good thing about CWA is that it guarantees the expert system to always have an answer; the bad thing is that it sometimes will provide an incorrect answer (when the difference between “I don’t know.” and “No.” matters).

Besides expert system, and similar usage such as automated answering machines, logic programming also have some other modern usages. One of them is automated theorem proving. Automated reasoning over mathematical proof was a major impetus for the development of computer science. [7] Here is a very simple example:

Suppose we have f(a), and we also have f(x) -> g(x). We want to know if g(a) is true. We can do this in logic programming, for example, in Prolog.

We first input those into the script:

```
f(a).
g(x):-f(x).
```

Then we input this as query:

```
?- g(a).
```

Prolog will tell us:

```
yes
```

The example may first seem useful only to mathematicians who can program; however it is actually very useful for computer scientists as well because now we have ways to prove whether a piece of code or the logic behind it is correct.

Why do we need to prove it to be correct? Can’t we just debug it? The answer is very simple: debugging can only show the presence of bugs, but not the absence. For any serious coding, for example, a program for flying an intercepting missile, building a bridge, or flying space shuttles, we need to prove that the logic is totally correct and bug free, instead of just testing it in run time for a while and saying “Well it seems to work, that’s it!” (just like what most game developers do, and oh those games are full of bugs!) As mentioned earlier, almost all errors and bugs in logic programming are caught during compile time instead of run time, so in addition to pure theories such as loop invariants and induction, now we have logic programming, an actual thing, instead of a theorem, as our tool to test whether our actual imperative language code is correct. So when we need it, we can say “We can prove that our program works.” instead of saying “It doesn’t have any bugs that I know of, but I’m not sure if it is bug free.”(another example of open-world-assumption!)

Dedicated Machine

There has been some attempts to design dedicated architectures for Prolog to help it run faster, such as “High Performance integrated Prolog processor, IPP”[3], "Pattern Addressable Memory (PAM)”[3], and "Prolog machine PSI (Personal Sequential Inference machine)”[3], etc.

Unfortunately, none of these dedicated designs were implemented for 2 main reasons. First, Prolog and other logic programming languages do not have enough impact on industry. Most programs written in Prolog are very small by industry standards (most of them are fewer than 100,000 lines of code). Prolog has compatibility issues with other languages, and it has portability issues as well. Most importantly, comparing with traditional languages, Prolog and other logic languages are often much slower than traditional languages. Although after decades of development and optimization, this margin of difference has been reduced, it’s still fairly obvious in most cases. Second, the need and growth of general-purpose hardware has consistently overtaken specialized hardwares, and since Prolog is not that popular in industry after all, no one bothered to produce actual machines specifically for it.[3] As a result, currently we don’t have any specialized machine for Prolog, or any other logic programming language.

Unification

Logic Variables

Logic Programming is a paradigm in programming where the programmer defines their code as a collection of clauses and facts. Once the clauses and facts are established, the programmer creates queries, which are run by an implication algorithm. The implication algorithm is able to use the clauses to determine if the query is true or false. Prolog programmers write out clauses in a special form of clause called a Horn clause, where one literal is implied by the other(s). Horn clauses are useful as they are easily understood by humans and make implicative reasoning easier for both human and machine.

Below are three examples of the same clause. In each case, the clause is only false if the object it applies to is an Apple, but is not a Fruit.

Basic Clause	Horn Clause	Prolog
¬Apple ∨ Fruit	Fruit ⟵ Apple	fruit(X) :- apple(X)

Facts are statements known to be true, but in logic, a fact can be thought of as a clause with a single literal which is true[13]. As an example of a fact, if John likes fruit, it could be written in Prolog as:

```
likes(john,fruit).
```

Logic variables are the traditional variables used in logic and mathematics to denote an unknown[13]. Unlike variables in imperative languages, logic variables do not denote some place in memory where values can be stored at any time. Logic variables are instead bound to other variables or specific values during unification[15]. If a logic variable A is bound to B, A becomes essentially an alias for B. This aliasing can be used to relate clauses containing A and clauses containing B, but the binding cannot be undone arbitrarily. [14] A is bound to B until a contradiction is found and the program backtracks past the point where the binding occurred.

Unification

Unification is a technique which takes a given equality, and joins the variables on either side so the equality holds[15]. If a contradiction is found during unification, then the equality given is not true. Several examples of unification are given below.

```
?- function(A,B) = function(1,2).
A = 1,
B = 2.
```

```
?- function(A,2) = function(1,B).
A = 1,
```

```
B = 2.
```

```
?- function(A,A) = function(1,2).
false.
```

In all three equalities, the function is the same. This must mean that its first argument on the left side of the equality matches the first argument on the right, and so forth. Assume for the sake of simplicity that A and B are unbound before the equality.

In the top equality, since the function is the same on both sides, there are two implications, A = 1, and B = 2. Since both are unbound, this means that A becomes bound to 1 and B becomes bound to 2. The center equality has the same function with the arguments in a different order. Its implications are that A = 1 and that 2 = B. This is logically identical to the left equality and has the same end result.

In the bottom equality, the implications are that A = 1, and A = 2. When the implication algorithm runs over this, it will try and bind A to one of the values, then the other. On the second binding, a contradiction occurs, 1 ≠ 2, so A cannot be bound to both values. This means that function(A,A) cannot be equal to function(1,2).

While binding a logic variable A to a value may look like assignment in other programming languages, they are distinct operations. Logic programming does not have assignment as its variables are not aliases for some memory, but instead aliases for some equality. If you assign A to B, then assign A to C, A will hold the same value as C, but there will be no relation between the three variables. However if you unify A and B, then unify A and C, all three variables will alias the same thing, either a value unified elsewhere with one of them, or an unknown.

Unification, while relating two clauses does not always evaluate them. Consider the following examples.

1	2	3	4
?- A = 1 + 3, | B = 2 + 2, | A = B. false.	?- A = 1 + 3, | B = 2 + 2, | A =:= B. A = 1+3, B = 2+2.	?- A = 3, | B = 1 + 2, | A is B. A = 3, B = 1+2.	?- A = 1 + 2, | B = 3, | A is B. false.

In example 1, a contradiction is found in the third clause, A = B. This may disappoint some first time logic programmers and make them wonder why it is called *logic* programming. This contradiction is warranted however, because A is bound to the *expression* 1+1, whereas B is bound to the *value* 2. The variable A cannot be unified with B because an expression is not a value[13]. In example 2, the "=:=" operator is used instead. This does not attempt to unify A and B, but instead evaluates both of them and checks if the results are equal.

Examples 3 and 4 both use "is" but have flipped input. The "is" operator does both unification and

evaluation. It evaluates the right side and unifies that result with the left side[13]. Because of this, example 3 is correct, but example 4 fails. In the third clause of example 3, B is first evaluated to 3. This is then unified with A, which is also 3, so it succeeds. In example 4, B is evaluated to 3, but then a contradiction is found when unifying it with A, since A is the expression 1 + 2. Through these four examples, we see that Unification is distinct from evaluation, though there are language features in Prolog to allow pure evaluation as well as partial evaluation with unification.

Pattern Matching versus Unification

Pattern Matching is a concept used in many languages to allow control flow to be directed based on what pattern input data conforms to. Many functional languages, beginning with Hope in the 1970's have included some form of pattern matching and some languages, such as AWK, are even built around it[16]. Languages such as Haskell, ML(and derivatives) and Curry have included pattern matching in the style of Hope. In many cases it allows for a high level of expressiveness, especially when combined with algebraic data types. Languages which support both features allow complex data structures like trees and lists to be created and manipulated with only a few lines of code.

Unification, however, is even more powerful than pattern matching. The paradigm of functional programming is to transform an input to an output, whereas logical programming is based around relating clauses in general. Unification does not involve an input and output, and is therefore not restricted to matching patterns in a single direction. This allows for code which would seem impossible to programmers unfamiliar with functional programming, like the following:

```
?- function(A,2) = function(1,B).
A = 1,
B = 2.
```

Because unification is not directional, it is able to obtain values for A and B regardless of the order of the equality given(as order doesn't matter) and is able to generate any value in a relation, as opposed to pattern matching only being able to generate the output.

The following three examples shows Prolog solving queries to an append relation. In a functional environment, the first two arguments would be input and the final argument an output. The three examples show solutions in a standard forward direction, as well as reverse direction, and exhaustively finding all inputs for a given output.

X as "output"	X as "input"	Exhaustively finding output
?- append([1],[2],X). X = [1, 2].	?- append([1],X,[1,2]). X = [2].	?- append(X,Y,[1,2]). X = [], Y = [1, 2] ; X = [1], Y = [2] ;

		X = [1, 2], Y = [] ; false.

In summary, though pattern matching and unification can be used to solve similar problems, unification does not have the restriction of solving in a single direction, and can instead solve for any combination of inputs so long as they cause no contradictions.

Free Variables and Unification in Other Languages

Though no non-logic-based languages directly support variable unification, it is possible to emulate the functionality in many languages. One particularly convincing example is in Tcl:

```
proc direction_adjective {dir adj} {
   set d_a {north boreal south austral west occidental east oriental}
   if {[set i [lsearch $d_a $adj]]>=0 && [regexp {^[A-Z_]} $dir]} {
               uplevel set $dir [lindex $d_a [incr i -1]]
   } elseif {[set j [lsearch $d_a $dir]]>=0 && [regexp {^[A-Z_]} $adj]} {
               uplevel set $adj [lindex $d_a [incr j]]
   } elseif {$j==$i-1} {return 1 } else {return 0}
}

-- Repl --

% direction_adjective D boreal
 north
% direction_adjective west A
 occidental
```

[8]

Though this example is interesting, it does not have the same declarative nature as Prolog. In the Prolog version the unification is a first class feature of the language. The Tcl version must specify how exactly the unification happens. It may be possible in Tcl to generalize this pattern into a library. Tcl supports higher order functions and lexical closures, so in theory one could create various functions for N-ary associations, that take in an association set and return a function with the the unification logic closed over that set. A similar approach to the one described was taken to add unification to haskell[9]. Additionally, a version of unification shows the NLTK (Natural Language Toolkit) for python[10].

```
direction_adjective(north, boreal).
direction_adjective(south, austral).
direction_adjective(east, oriental).
direction_adjective(west, occidental).

-- Repl --

? direction_adjective(D, boreal).
 D = north
? direction_adjective(west, A).
 A = occidental
```

Curry, a functional logic programming language, supports the unification of logic variables. There are several important differences between Curry and Prolog. The first big difference is the way a relationship is setup in Curry. Unlike Prolog, a relationship in Curry must explicitly set to True. This is necessary because of Curry's functional roots. Another difference is that all free variables must be declared in in Curry. In Prolog, all words starting with capital letters are going to be free variables. This is not possible in Curry because of it's adoption of Haskell's naming conventions, where data structures and data constructors start with capitals, and functions start with lowercase. This convention is also important to avoid accidentally creating a function with the same name as a free variable used elsewhere in the program.

```
data Direction = North | South | East | West

data Climate = Boreal | Austral | Oriental | Occidental

directionAdjective :: Direction -> Climate -> Bool
directionAdjective North Boreal    = True
directionAdjective South Austral   = True
directionAdjective East Oriental   = True
directionAdjective West Occidental = True

-- Repl --

? directionAdjective d Boreal where d free
 d = North
? directionAdjective West a where a free
 a = Occidental
```

As a slightly more complicated example of free variables and unification mixed in with a functional language can be explored by examining this implementation of append:

```
append' []     ys = ys
append' (x:xs) ys = x : append' xs ys

-- Repl --

? append' [1,2] [3]
| [1,2,3]
? append' b a =:= [1,2,3] where a, b free
| a = [], b = [1,2,3]
| a = [1], b = [2,3]
| a = [1,2], b = [3]
| a = [1,2,3], b = []
? append' [1,2] a =:= [1,2,3] where a free
| a = [3]
```

Non-Determinism and Backtracking

While Prolog itself is nondeterministic in that at any point there can be more than one branch that can be taken and still be legal, implementations of it usually deterministic. Having non-determinism in an implementation of a language can make it very difficult to reason about a program's behaviour and make testing next to impossible, even though prolog is written by stating out the rules which largely removes the need for explicit testing. This wouldn't take into account the programer making incorrect reasonings about the rules he provides prolog, but then again there's nothing stopping him or her from doing the same thing when writing the tests for a normal deterministic language[17].

The way non-determinism in Prolog works is similar to how there are many legal depth first searches of a graph or tree when deg(v)>2 for some v∈V(G)\{root}, or if deg(root) > 1. You can however make it deterministic by writing some kind of rule for which order the adjacent edges of the current vertex would be DFS'd in, for example in ascending order. Some implementations of Prolog do this for the backtracking searches by imposing a rule that clauses will be evaluated in the left-to-right order where they are written.

Unlike Prolog the programming language Curry is able to be nondeterministic in the sense that any function can have multiple definitions. If a function has more than one definition then the programer will be unable to predict the outcome when that function is used. Curry's nondeterminism can be good in a situation where randomness is wanted, like when making random permutations[18]. Here is an example of nondeterminism:

```
coin = 0
coin = 1
double x = x + x
```

The value of x will either be 0 or 2 since the value of the two x's on the right hand side are bound to each other and the selection of which coin to use is nondeterministic.

Prolog's backtracking mechanism is a very important performance improvement because any choice that could not lead to correct matching to the desired predicate is immediately halted. The mechanism works by moving the current state of the search back to the last state whenever a situation is reached where the predicate cannot evaluate to true. When Prolog attempts to match the desired predicate it will start with the first value, then the second value, and onwards, until it reaches a correct match or Prolog decides to backtrack which will make it go back and try a new value in the current argument. An operator that is offered by Prolog is Cut (!) which allows the programmer to decide how many of the arguments will be iterated through, while the rest are just stay the same throughout the whole search[19]. for example:

```
a(X, Y) :- b(X), !, c(Y).
b(1).
b(2).
b(3).

c(1).
c(2).
c(3).
```

If the user tried to search for some a(X, Y) from the above code the only value of X that would be used is 1 since the cut operator is right after b(X) in the definition of a(X, Y).
There are two different ways you can categorize cuts into two different categories: green cuts and red cuts. Green cuts do not affect the behavior of the program only the efficiency of it, whereas red cuts also affects the behavior of the program.[20]

Writing some things in Prolog is very easy since you only need to write down what you want the answer to be, whereas in an imperative language you would have to write down all the steps to getting there. Here is an example, Magic Squares:

```
d(1).
d(2).
d(3).
d(4).
d(5).
d(6).
d(7).
d(8).
d(9).

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

magic(A,B,C,D,E,F,G,H,I) :- d(A), d(B), d(C),d(D), d(E), d(F), d(G), d(H), d(I),
                            check(A,B,C), check(D,E,F), check(G,H,I),
                            check(A,D,G), check(B,E,H), check(C,F,I),
                            check(A,E,I), check(C,E,G).
```

Writing a similar problem in another language, say Python/Java/C++/etc would take a bit longer as you would have to explicitly write in the code that checks all possible inputs and the recursive call if you wanted to implement it as some kind of backtracking algorithm. The recursion makes the backtracking a lot more natural to program as when you need to backtrack up the tree when all options at the current branch have failed, you simply return from the call to its callee. The (incredibly inefficient) solution that wouldn't use backtracking would simply be 9 nested loops that try every single possible permutation of the cells, which would result in 9! (or in general, n! for an n-cell square) attempted permutations. If it was to be made general you would have to use recursion since you couldn't do n nested loops otherwise.

Datalog and Relational Databases

Introduction of Datalog

Datalog is a determination model of logic programming and it is a typical programming language, Prolog,to the area of databases. The thing that makes logic programming being very appealing is its rhetorical nature, as it is opposed to the idea of having more operational characteristic of other programming models that is being involuntary, object-oriented, or functional. [25]

Datalog expand connective queries with recursion that is to uphold such types of queries. A Datalog program is supported by a set of rules, and each of these rules is a connective query. Furthermore, recursion is inferred and used by permitting the exact relational symbols in both the heads and the bodies of the rules. The fundamental constraint that differentiates Datalog from Prolog is the fact that it is not allowed to use function symbols. [25]

The package of Datalog has a lightweight constructive database system. Datalog is used for Queries and databases updates and that is a declarative logic language where in each formula is a function-free Horn clause, and for each variable in the head of a clause must show in the body of the clause. The use of Datalog syntax and implementation are based on tabling intermediate results, which ensures that all queries terminate. [26]

Syntax of Datalog

The syntax of Datalog is inherited from the logic programming language Prolog. Also, it has proviso that allows only constants and relational symbols (no function symbols). [25]

A Datalog rule has the form: T(x) :- q(x , y)

where x = x1, . . . , xn is a tuple of known variables, y = y1, . . . , ym is a tuple of holding a large number of variables. T is a relation and q is a unification of relational atoms. As depicted from the left-hand side, it is called the head of the rule and it work in with the output and result of the query. Moreover, the right-hand side is called the body of the rule. Notice that all known variables in the head, it needs to show off at least one atom in the body. [25]

A Datalog program is a finite set of Datalog rules over the equivalent schema. Relation symbols are called predicates. There are two types of the and one of them is shown only in the body of the program's rules and they are called extensional database predicates. In the mean time, those that appear in the head of some rule are called intensional database predicates. A Datalog program expresses a Datalog query when one of the intensional database predicates is specified as the output. [25]

In the matter of Datalog input, whitespace characters are ignored unless if they disconnect neighbouring tokens or in the case when they appear in strings. In addition, comments are considered to be white space, unless the user use the character '%' which introduces a comment and it extends to the next line break. However, comments do not happen inside strings. [26]

The characters in Datalog input are assembled into tokens which keeps track of the rules that follow accordingly. The following four classes of tokens are: punctuation, variables, identifiers, and strings. [26]

The punctuation tokens are depicted as follows: ‘(’, ‘,’, ‘)’, ‘=’, ‘:-’, ‘.’, ‘~’, ‘?’, and ‘"’.
A variable is a range of Latin capital and small letters, digits, and the underscore character. A Latin capital letter must be at the beginning of a variable. [26]

An identifier is a range of printing characters which does not include any of the following characters: ‘(’, ‘,’, ‘)’, ‘=’, ‘:’, ‘.’, ‘~’, ‘?’, ‘"’, ‘%’, and space. When using an identifier, it must not start with a Latin capital letter. However, while hyphen character is allowed, the characters that begin with punctuation are forbidden in identifiers. [26]

A string is a range of characters circumfluent in double quotes. Characters can be directly included in a string such as other than double quote, newline, and backslash. However, the remaining characters can be specified using escape characters, ‘\"’, ‘\n’, and ‘\\’ accordingly. [26]

Benefits of Datalog

Much like other query languages such as SQL, Datalog is declarative. In other words, a datalog program describes what computation should be performed and not how to perform a computation. This allows the programmer to state the logic of an algorithm and not necessarily the control. Datalog uses declarative programming methods for expressing distributed states and creating strategies for computing and maintaining these states. Each Datalog program can be imagined as a recursive specification of its intensional database predicates. This specification is expressed using equations involving the unions of conjunctive queries.

Example:

Datalog program

```
T(x,y) :- E(x,y)
T(x,y):- T(x,z), T(z,y)
```

Recursive equation:

$$T = E \cup \pi_{1,4}(\sigma_{\$2=\$3}(T \times T))$$

The smallest solution of this recursive equation is the Transitive Closure of E

Classic example of the declarative semantics of Datalog programs.[30]

Simplicity

Datalog as a declarative program achieves a high level of simplicity. The basic element of Datalog is a clause, which is essentially a list that either begins with the name of rule, or is a data pattern. Variables can be embedded into these clauses (symbols that start with '?'). At computation, the query engine finds all the available combination of variables that correspond to the clauses. [31]

The essence of datalog is the ability to define recursive relations. Relations are the main datalog type and computation involves using a set of inputted relations to infer the contents of all relations. Datalog simplifies writing recursive queries and also makes query optimization easier. Recursive rules in Datalog

have the same predicates in the head and body of the rule. Recursion is very important to Datalog because without it, Datalog can express only the queries of relational algebra.
Some of the ways to express the recursive semantics of Datalog programs include:

1. **Minimal relations (minimal models):** Logical descriptions of new relations.
   Example:

```
• A program
    Ancestor(x,y) ← Parent(x,y)
    Ancestor(x,y) ← Parent(x,z) AND Ancestor(z,y)
• corresponds to logical properties
        P1 ∀x ∀y (Parent(x,y) → Ancestor(x,y))
        P2 ∀x ∀y ∀z(Parent(x,z) & Ancestor(z,y) → Ancestor(x,y)).
```

2 **Provability semantics:** proof theoretic way of considering Datalog programs

3 **Fixpoint semantics: E**xecuting rules until no new tuples are added (i.e a fixpoint is reached). If the ruler are safe, finitely many tuples will satisfy the body of the rule, and as such a fixpoint is reached after a finite number of rounds. [32]

Example: fixpoint construction

```
 Ancestor(x,y) ← Parent(x,y)
 Ancestor(x,y) ← Parent(x,z), Ancestor(z,y)
     • Start: Ancestor = {}, Parent={<a,b>,<b,c>,<c,d>}
     • 1st round: Ancestor = {<a,b>,<b,c>,<c,d>}
     • 2nd round: Ancestor = {<a,b>,<b,c>,<c,d>, <a,c>, <b,d>}
(Ancestor(a,c) ← Parent(a,b), Ancestor(b,c) gives <a,c>
 Ancestor(b,d) ← Parent(b,c), Ancestor(c,d) gives <b,d>)
     • 3rd round: Ancestor = {<a,b>,<b,c>,<c,d>, <a,c>,<b,d>,
 Ancestor(a,d) ← Parent(a,b), Ancestor(b,d) <a,d>}
     • 4th round: no new tuples in Ancestor.
```

Uses of Datalog

Datalog ideas and algorithms are widely used in different database systems. In Python, there is a library that uses Datalog and its called pyDatalog. It basically adds logic programming to python's toolbox. It can operate logic queries on databases or python objects, and use logic clauses to express the behavior of python classes. It supplements Python for the following manners:

- ❖ querying complex sets of related information
- ❖ demonstrate smart manner and this is usually needed for games or expert systems
- ❖ execute recursive algorithms and this is neede in network protocol, code and graph analysis

Datalog is considered to be outstanding at directing complexity. most of the time Datalog are shorter than their Python valent, nevertheless, its statements can be specified in any order. [28]
pyDatalog provides interesting features that enable users to do useful operation with queries and database implementation. Some of the following features are:

- It gives the user the ability to assert facts in a datalog knowledge base, and query it.
- It provides the advantage of using logic clauses to query any relational database via SQLAlchemy, the data access layer for Python which supports 11 dialects of SQL.
- The user can define attributes of Python classes through logic clauses, and use logic queries on Python objects.
- It supports the use of interactive console to run datalog queries.

pyDatalog is considered to be fast in speed and lightweight datalog interpreter that is written in Python, and the depth of recursion is not bounded by the size of stack. [28]

Prolog and Datalog

Prolog is a general purpose logic programming language. It is considered to be the most popular logic programming out there. When Prolog was in its design phase, it was expected that it will be used as a database language, however, some of Prolog's feature were not suitable to a database application, which was the motivation of creating Datalog, a subset of Prolog that will be a database and logic programming language.

Syntactically, Datalog is very similar to Prolog. It is even possible to parse and execute Datalog clauses in a Prolog Interpreter. However, Datalog and Prolog differ in semantics, in particular by the specification of how Prolog and Datalog must execute their programs [21]

A Prolog program is processed and executed according to Prolog's strategy specification, that is, it uses a depth-first-search method to construct a tree with backtracking. Prolog scans clauses from top to bottom with the first clause being the leftmost branch to be resolved, hence the order of the clauses and literals plays a big role in Prolog. Prolog also uses backtracking to keep track of the points of the tree where there could be multiple choices (i.e splitting into separate branches). Prolog is known to be based on this principle, which is known as backward chaining. However, this principle might sometimes leads to situations where Prolog programs go into an infinite loop. An important difference between Prolog and Datalog is that order of the of the clauses and literals do not matter, what so ever, in Datalog. Prolog processes one fact at a time, whereas Datalog processes sets of facts at a time. [22]

Another difference between Datalog and Prolog is that in Datalog, the user cannot create a complex predicate. Every predicate consist of one or zero variable or constant, i.e, Datalog does not allow for a function term to be an argument of the predicate [23] As mentioned earlier, Datalog syntax consists of head and a body.

$$Body_1, ..., Body_n :- Head$$

If $n = 0$, Datalog considers it to be a fact, or extensional database (EDB), otherwise, it is considered a rule, or intentional database (IDB). “The EDB predicates value are given via an input database, while the IDB predicates value are computed by the program. In standard Datalog, EDB predicate symbols may only appear in Body (Cal et al., 2010). In other words, those predicates appearing in head of a non-zero predicate

in the body are all IDBs." [23]

The size of the EDB predicates plays a big role in the complexity of computing whether a literal in a Datalog program is from an EDB or an IDB [24]

Because Datalog is designed to be a database language, its semantics must differ from Prolog to make it more suitable for database applications. Prolog's semantics are defined by operational semantics, whereas Datalog only uses declarative semantics. Also, when querying a Prolog program, Prolog returns its results in one tuple at a time, (each of these tuples returned True in the proof tree constructed by Prolog), however, Datalog uses a set-oriented data processing strategy to return its results. [21]

Safety rules

Safety is important in Datalog as it won't make sense to write some rules that could possibly generate an infinite amount of answers, such as:

- gt(Z,Y) :- Z> Y
- re(X) :- **not** X

Thus, Datalog has some safety conditions that needs to be satisfied:

1. "Every variable that appears in the head of the rule also appears in a non-arithmetic positive literal in the body of the rule" , so in our first example, we cannot have Z>Y in the body since this in an arithmetic form.
2. "Every variable appearing in a negative literal in the body of the rule also appears in some positive literal in the body of the rule". In our second example, the variable X appeared in a negative form in the body, but there are no other rules that have X in its body in a positive form

A non-recursive Datalog program that satisfies these safety conditions will guarantee that the view relations defined in the program will be finite. [27]

Relational Algebra and Datalog

Here are some interesting examples from [29] on Relation Algebra and Datalog that shows how Datalog is simple and easy to follow.

Schemas used in the examples

```
R(name, address, gender, birthdate)

S(name, address, gender, birthdata)

Movie(title, year, length, studioName)
```

1. Intersection

Relational Algebra	Datalog
I := R ∩ S	U(n,a,g,b) :- R(n,a,g,b) AND S(n,a,g,b)

2. Union

Relational Algebra	Datalog
I := R ∪ S	U(n,a,g,b) :- R(n,a,g,b) U(n,a,g,b) :- S(n,a,g,b)

3. Difference

Relational Algebra	Datalog
D:= R - S	D(n,a,g,b) :- R(n,a,g,b) AND NOT S(n,a,g,b)

4. Projection

Relational Algebra	Datalog
P:= π title, year, length(Movie)	P(t,y,l) :- Movie(t,y,l,s)

5. Selection

Relational Algebra	Datalog
S:= σ length >= 100 AND studioName = 'Fox'(Movie)	S(t,y,l,s) :- Movie(t,y,l,s) AND l>=100 AND s = 'Fox'

REFERENCES

[1]The Birth of Prolog (November 1992) Alain Colmerauer and Philippe Roussel [Online] Viewed 2013 November 27: http://web.archive.org/web/20070703003934/www.lim.univ-mrs.fr/~colmer/ArchivesPublications/HistoirePrologue/19november92.pdf

[2]Introduction to Prolog Programming (February 2012) Bill Wilson [Online] Viewed 2013 November 27: http://www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/intro.html

[3] Prolog (November 2013) Wikipedia [Online] Viewed 2013 November 27: http://en.wikipedia.org/wiki/Prolog

[4] Logic Programming with Prolog (2012) University of Birmingham [Online] Viewed 2013 November 27: http://www.cs.bham.ac.uk/~pjh/prolog_course/sem223_se.html

[5]Strengths and Weaknesses of Logic Programming (2005) UNSW Australia [Online] Viewed 2013 November 27: http://seit.unsw.adfa.edu.au/coursework/ZEIT3113/lectures/05/PL5/PL5.htm

[6] Expert System (November 2013) Wikipedia [Online] Viewed 2013 November 27: http://en.wikipedia.org/wiki/Expert_system

[7] Automated Theorem Proving (October 2013) Wikipedia [Online] Viewed 2013 November 27: http://en.wikipedia.org/wiki/Automated_theorem_proving

[8] Playing Prolog (Dec 2000). Tcl Tk [Online] Viewed 2013 November 27 Available: http://www2.tcl.tk/755

[9] Control.Unification (Undated). Haskell.org [Online] Viewed 2013 November 27 Available: http://hackage.haskell.org/package/unification-fd-0.5.0/docs/Control-Unification.html

[10] Feature Structure & Unification (Undated). Natural Language Toolkit [Online] Viewed 2013 November 27 Available: http://nltk.googlecode.com/svn-/trunk/doc/howto/featstruct.html

[11] Case Expression and Pattern Matching (March 1997). A Gentle Introduction to Haskell[Online] Viewed 2013 November 27 Available: http://www.cs.auckland.ac.nz/references/haskell/haskell-intro-html/patterns.html

[12] Overlapping Rules and Logic Variables in Functional Logic Programs (September 2006). International Conference on Logic Programming[Online] Viewed 2013 November 27 Available: http://www.informatik.uni-kiel.de/~mh/papers/ICLP06.html

[13] The Prolog Dictionary (July 2013). University of New South Wales Computer Science and Engineering [Online] Viewed 2013 November 27 Available: http://www.cse.unsw.edu.au/~billw/Prologdict.html

[14] Unification and Logic Variables (Undated). Eclipse Tutorial Introduction [Online] Viewed 2013 November 27 Available: http://eclipseclp.org/doc/tutorial/tutorial015.html

[15] Unification: Pattern Matching but twice as nice! (May 2011). LShift [Online] Viewed 2013 November 27 Available: http://www.lshift.net/blog/2011/05/31/unification-pattern-matching-but-twice-as-nice

[16] Pattern Matching (Undated). A Hope Tutorial [Online] Viewed 2013 November 27 Available http://www.soi.city.ac.uk/~ross/Hope/hope_tut/node13.html

[17] Incremental search and non-determinism [Online] viewed 2013 November 27: http://prolog.cs.vu.nl/pldoc/doc_for?object=section(3,'7.1',swi('/doc/packages/space.html'))

[18] Programming with narrowing: A tutorial [Online] viewed 2013 November 27: http://web.cecs.pdx.edu/~antoy/homepage/publications/narrowing/paper.pdf

[19] Cuts in Prolog [Online] viewed 2013 November 27: http://cs.union.edu/~striegnk/learn-prolog-now/html/node88.html

[20] Cut (logic programming) viewed 2013 November 27: http://en.wikipedia.org/wiki/Cut_%28logic_programming%29

[21] What you always wanted to know about Datalog (and never dared to ask) (2002) [Online] Viewed 2013 November 27:

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=43410&tag=1
[22] T. Lu, "Compilation and Evaluation of Nested Linear Recursions: A Deductive Database Approach" M.Sc Thesis, Simon Fraser University, Vancouver, BC, Canada, 1993:
http://summit.sfu.ca/system/files/iritems1/5746/b15249761.pdf
[23] Query Evaluation of Tractable Answering using Query Rewriting (2012) Zakka Muhammed [Online] Viewed 2013, November 27:
www.emcl-study.eu/fileadmin/master_theses/thesis_muhammad.pdf
[24] Complexity and Expressive Power of Logic Programming (2001) Evgeny Danstin, Thomas Eiter, etal [Online] Viewed 2013, November 27:
http://cmpe.emu.edu.tr/bayram/courses/531/ForPresentation/p374-dantsin.pdf
[25] DATALOG ICS [Online] Viewed 2013 November 27:
www.ics.forth.gr/~gregkar/publications/encyclopedia-datalog.pdf
[26] Datalog User Manual (2004) John D. Ramsdell [Online] Viewed 2013 November 27:
http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html
[27] Datalog, SFU [Online] Viewd 2013 November 27:
www.cs.sfu.ca/CC/354/jpei/slides/Datalog.pdf
[28] pyDatalog(April 2013) Google [Online] Viewed 2013 November 27:
https://sites.google.com/site/pydatalog/
[29] Database Systems: Logical Query Languages, University of Illinois at Urbana-Champaign, [Online] Viewed 2013 November 27:
www.cs.uiuc.edu/class/fa07/cs411/lectures/cs411-f07-datalog.pdf
[30] Alechina, N , "Theory of Relational Databases". Retrieved11 , 2013 Available:
www.cs.nott.ac.uk/~nza/G53RDB07/rdb14.pdf
[31] Hickey, R , "Datomic Queries and Rules". Retrieved11 , 2013 Available:
http://docs.datomic.com/query.html
[32] Senz Prez, F , "A Deductive Database with Datalog and SQL Query Languages". Retrieved11 , 2013 Available: www.fdi.ucm.es/profesor/fernan/fsp/scg11aslides.pdf