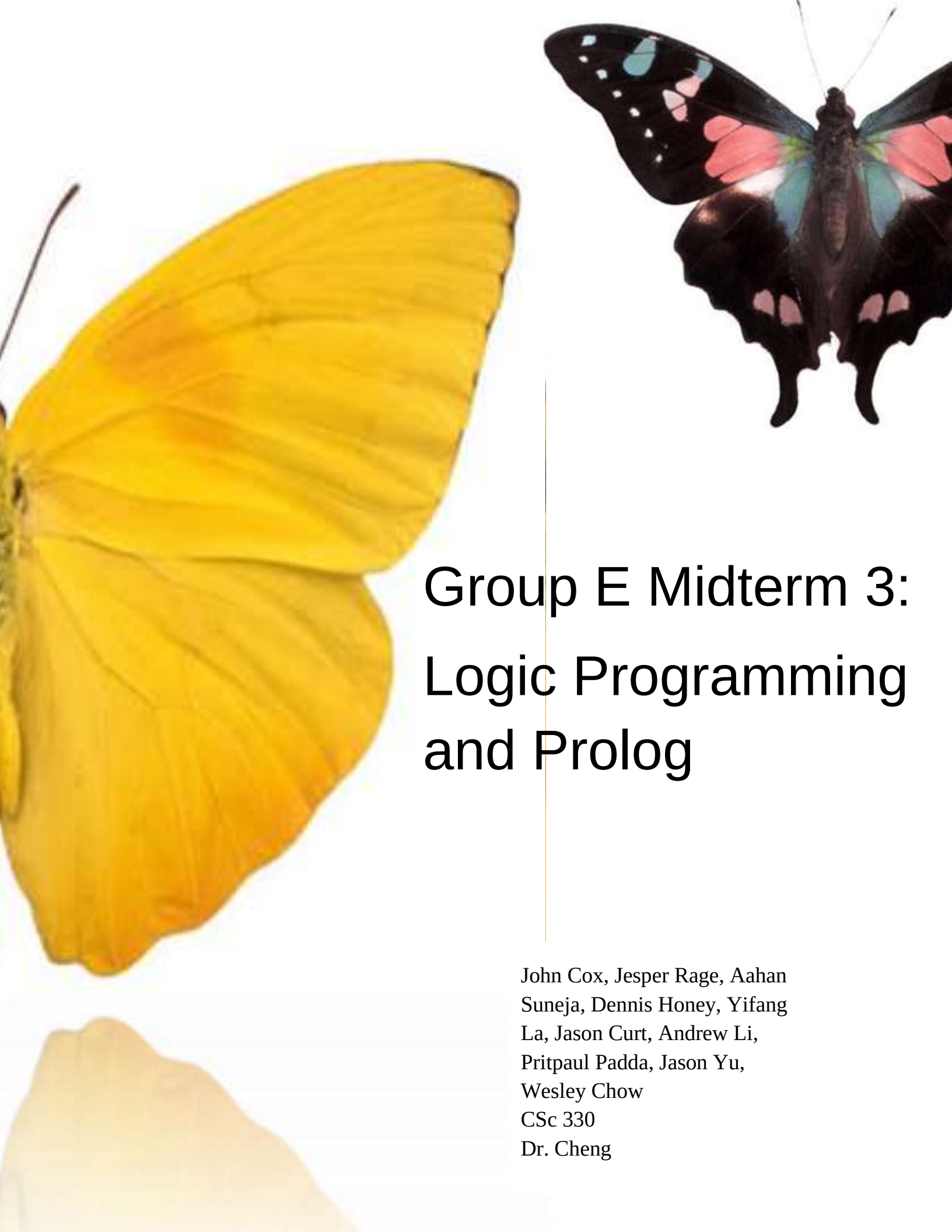


Group E Midterm 3:

Logic Programming and Prolog

John Cox, Jesper Rage, Aahan Suneja, Dennis Honey, Yifang La, Jason Curt, Andrew Li, Pritpaul Padda, Jason Yu, Wesley Chow
CSc 330
Dr. Cheng

TABLE OF CONTENTS

SUMMARY

The purpose of this report is to research and demonstrate our understanding of logic programming and Prolog. We will do this by covering four topics: History or Prolog; unification and logic variables; non-determinism and backtracking and datalog and relation databases.

This report was researched by ten group members over a seven hour open book exam. The information for the report was gathered using the internet and search engines.

1. HISTORY OF PROLOG

INTRODUCTION

PROLOG (PROgramming in LOGic) was first conceived by Alain Colmerauer in Marseille, France and Robert A. Kowalski in London, Britain, in the early 1970s[1]. In 1972, Alain Colmerauer and Philippe Roussel developed the first PROLOG system. PROLOG is a general purpose programming language that is declarative, which is a programming paradigm that expresses the logic of computation without describing the control flows [2]. The program logic is expressed in terms in relations, represented as facts and rules, which are also known as rules of inference or transformation rules. These rules are acts of drawing a conclusion based on the given statements, which are interpreted as a function that takes other statements; this function also analyses the statements' syntax and returns a conclusion. In order to initiate a computation, a query must be run over the relations of the program logic. PROLOG was invented during a research project that was attempting to process natural languages (French), but resulted in producing PROLOG [3].

EARLY APPLICATIONS

The first implementation of Prolog was an interpreter written in Fortran by members
of Alain Colmerauer's group[26]. It was developed in 1972 by Colmerauer with Philippe Roussel and Kowalski

An early application of Prolog is natural language processing, which is the interaction between human natural languages and computers. The language itself is used to implement many processes found in it but the particular example deals with finite state automatas:

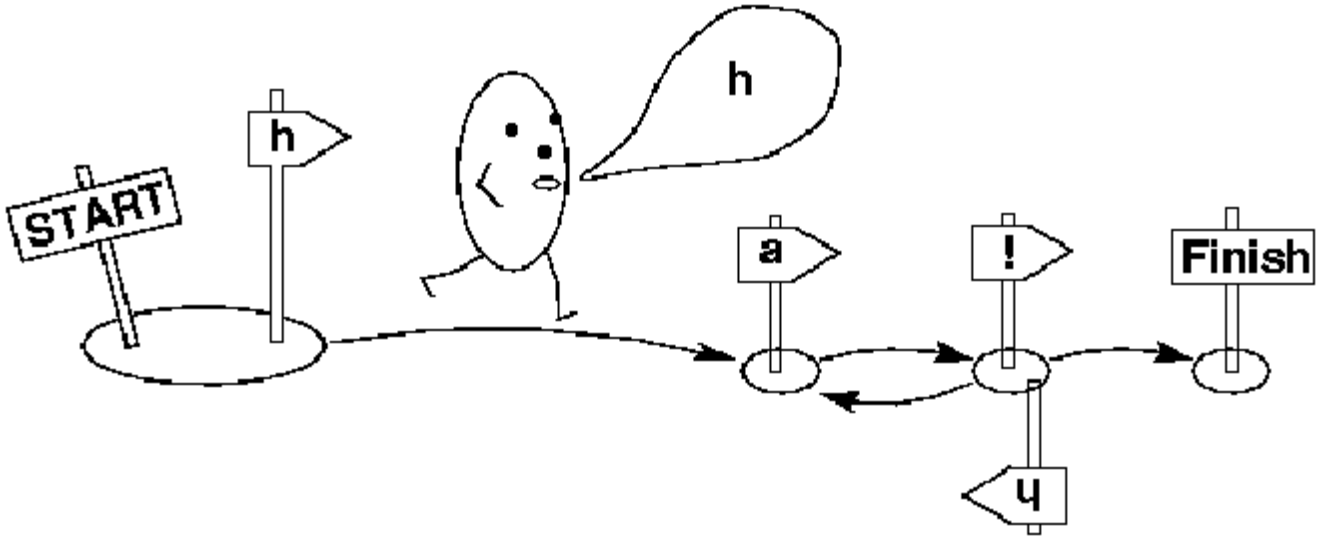


Figure: The finite state machine. The states are the paths while the walking egg is the program going through the FSM.

To generate such code to represent the finite state machine, the code is below[28]:

```
--
initial(1).
final(4).
arc(1,2,h).
arc(2,3,a).
arc(3,4,!).
arc(2,1,a).
--
```

Compare this to a typical implementation in an object-oriented program; it would take longer to create the same finite state machine and figure out the logic as you cannot define transition states as easily as a developer can in Prolog. As the implementation of this FSM is a passive data structure (the declarative part), the procedural part can be added in addition to the declarative part, which is analogous to the egg in the diagram.

[25] Some applications of Prolog are:

- Intelligent Database Retrieval

- Natural language Understanding
- Expert Systems
- Specification Language
- Machine Learning
- Robot Planning
- Automated Reasoning
- Problem Solving

RECENT APPLICATION

Logic programming is best suited for constraint satisfaction problems, but it is not the only thing it's good for. Logic programming has many applications areas:

- Natural Language Interfaces
- Expert Systems
- Symbolic Equation solving
- Planning
- Prototyping
- Simulation
- Programming Language Implementation

A recent application of logic programming was in IBM's Watson System, which defeated former Jeopardy! Grand champions, Ken Jennings and Brad Rutter. Watson uses PROLOG for its Natural Language Processing (NLP) technology. NLP was used to analyse the high amounts of unstructured texts that may provide evidence in support of answers to the questions that Watson had to answer. Watson applies numerous detection rules that match patterns in many elements such as the focus of the question, the lexical answer type and the relationships between entities in either a question or a supporting passage. PROLOG was used to implement these rules because IBM needed a language that they could conveniently express pattern matching rules over the numerous parse trees and other annotations that Watson used.

Here is an example of how Watson attempts to process the following Jeopardy! Question:

POETS & POETRY: He was a bank clerk in the Yukon before he published "Songs of a Sourdough" in 1907.

Watson's NLP begins by applying a parser that converts the statement into a more structured form: a tree that shows both the surface structure and deep, logic structure. The output of the parser includes keywords and phrases like "published", "subject", "he", "Songs of a Sourdough" and etc.

Watson then applies the numerous detection rules that match patterns in the parser output, which are converted into the following PROLOG facts:

```
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).
```

These facts are then inputted into a PROLOG system and several rule sets are executed to detect the focus of the question, the lexical answer type and several relations between the elements of the parser output. A 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",...]).
```

The author and composition predicates apply constraints on the facts (“he” and “Songs of a Sourdough”) to rule out facts that aren't valid for the author and composition roles of the relation. This rule results in the new fact authorOf(1, 3), which is recorded and delivered to the downstream components of Watson’s pipeline [4].

PROLOG MACHINES

David H. D. Warren designed an abstract machine in 1983 for the execution of Prolog consisting of a memory architecture and an instruction set. Warren Abstract Machine is an abstract non-physical computer that helps in the implementation and compilation of the Prolog programming language[27]. It is similar in terms to the abstract machine based on Warren’s early work on the Prolog compiler (DEC-10). WAM offers procedures for optimizing and compiling symbolic computing that can be generalized beyond Prolog. David Warren’s main purpose of designing WAM is to compile Prolog code to a more low-level WAM code. David’s

reasoning to compile Prolog code to a lower-lever WAM code is to make the subsequent interpretation of the Prolog program more efficient. This is due to Prolog code being easy to translate to WAM instructions, which has the potential to be interpreted efficiently.

Another machine implemented other than WAM is the Vienna Abstract Machine (VAM). VAM is a Prolog machine developed at the Vienna University of Technology. Different from WAM, an inference in VAM is performed by unifying the goal and head immediately, instead of bypassing arguments through a register interface. Unique to WAM instructions, VAM instructions can be understood only by their combination at runtime.[27]

2. UNIFICATION AND LOGIC VARIABLES

UNIFICATION

Unification is the simple concept of finding two expressions that are equal, or "unified". The simple concept of unification can become quite complicated.

In Prolog there are only three kinds of terms: constants, variables and complex terms.
Constants are numbers like 54 or words with lowercase letters. Variables are words starting with uppercase letters. Complex terms are functions or terms with rules sets [5].

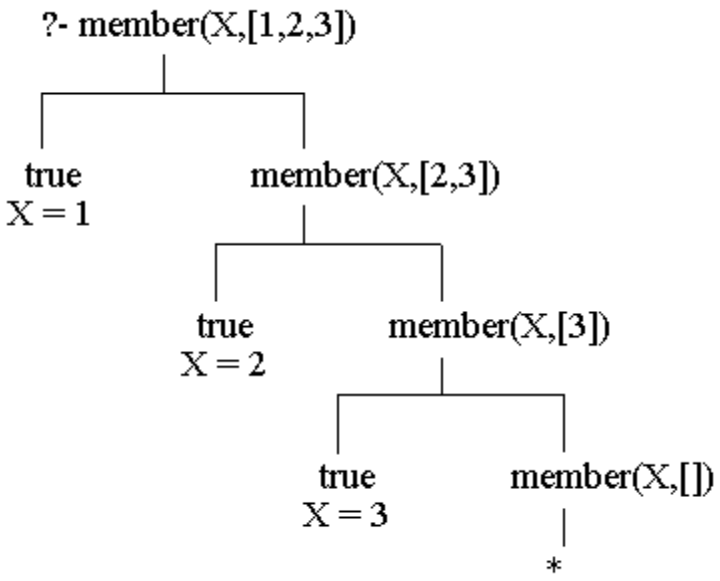
To unify two terms they must be "matched" and there are two cases where terms are considered matched. The first case is if the two terms are the same, the second case is if the terms contain variables that can be unified so that the two terms without variables are the same. These unifications are solved in a depth first search manner, that is each branch is fully traversed until either the conditions match or the conditions fail, when the conditions fail Prolog backtracks up the nearest branch and then traverse down the next branch [5].

It should be noted that all computation in Prolog is actually centered on unification.

Examples:

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R).
```

Prolog Unification for: ?= member(X,[1,2,3]) will look like this



[reference here]
On the left branches Prolog's unification has matched the first clause with member, and the right branches are matches against the second clauses[6].

PATERN MATCHING

Pattern matching attempts to determine the proper variant of a specified function for a certain instantiation. The given interpreter chooses a particular function based on the arguments that were passed. A function may be defined by multiple separate clauses which are each checked in turn. The search for a working clause ends after the first success. The list of patterns which are matched are ordered. Each clause may be disjoint from the other[10].

Examples:

```
last([LastItem], LastItem).
last([Head|Tail], LastItem) :- last(Tail, LastItem).
```

The result obtained is the last element of the list.

UNIFICATION VS PATTERN MATCHING

Unification resembles a bidirectional pattern matching logic. Pattern matching only allows variables on one side of the evaluator. For example in Prolog the statements

```
X = Y,
X = 5,
```
concludes with Y=5 as well [12].
Unification generalizes pattern matching. Pattern matching usually maps from a general expression to a specific instance. Unification allows variables in both expressions and assignments on either side. One important distinction is that in other methods such as Haskell, a clause is attempted to pattern match only until it finds the first success. In Prolog a predicate may be matched for every clause resulting in success. This search may be stopped short by a cut, but that is out of the scope of this discussion.

There are examples which fail for pattern matching but result in success from a unification standpoint:

```
(unify '[?a ?b] '[1 ?a])
;=> {?b 1, ?a 1}
```
This would not work when pattern matching [11].

The power of the unification algorithm shines when there are variable expressions on both sides of the evaluator.

UNIFICATION IN OTHER LANGUAGES

There is at least one non - Logic language that uses unification, its called Erlang! Erlang is a functional language that uses single assignment, dynamic typing and eager evaluation. Erlang was developed by Ericsson with the specific intention of simplifying concurrent programming and was released as open source in 1998. Erlang was based off Prolog and Haskell.

The simplest way to show how Erlang works is with examples, consider these lines to be executed one after another.

A = 1.	This is Assignment, A does not have a value so it is assigned the value
A = 1.	Because we already assigned a value to A it now uses Unification
A = A.	This is Unification, it will try solving this to true
1 = A.	This is Unification, the value of 1 is known, try to find an A that is valued at 1

A = 2. Will throw an error because its already evaluated A to the value 1

[7].
Others to note:

Haskell uses unification for type inference. There is a javascript library called JUnify which is used for unification of objects and arrays [8]. There are several so called functional logic programming languages like Mercury use unification to some degree or other. There are several languages based off Prolog with some level of unification. Almost all logic programming language uses unification.

3. NON-DETERMINISM AND BACKTRACKING

INTRODUCTION

Before introducing the concept of non-determinism in Prolog, it is important to briefly understand what it is. Simply put, a non-deterministic language is a language that contains specific points in computation called choice points. These choice points can allow the program to branch off into many different flows of simultaneous computation. On the other hand, in a deterministic language, such as Java, the flow of computation follows a specific and non-flexible path. Another way to think of non-deterministic flow, unlike if-then statements, is that the method of choice between branches of data flow are not determined by programmers and are instead decided by the program itself during run time. In languages, such as Prolog, a hierarchy of choice points will be made and higher tiered choices will branch into lower level choice points until a solution is found. If a solution is not found, it will go back to the next lowest hierarchy to explore other branches of computation. Using depth-first search, the program will run until a solution is found or all possible branches are explored.[17]

Here are some specific points that non-determinism and backtracking work in Prolog:

- When presented with several branches from a choice point, Prolog generates them one at a time.
- Since Prolog uses depth-first search, it will remain committed to a computation path until it succeeds or fails finitely.
- When a computation path terminates, it will backtrack to the most recent choice point with remaining untried branches.
- The program terminates if no solution is found after all the choice points and branches have been exhausted. The program will also “end” if a solution is found in one branch of computation, but will backtrack if the user requests another

solution. If all possible solutions are exhausted, the program will terminate.

- The order in which the branches or clauses are tried are based on their textual order within the program.[18]

N-QUEEN PROBLEM

Prolog has built in backtracking, and can easily solve problems with non-determinism; such as, the N-Queen problem. The N-Queen problem is how many queens can you fit on a NxN chess board without them being able to attack each other.
The following Prolog code shows a solution to the Queen Problem for an 8x8 board:

```
queens([]).   % when place queen in empty list, solution found
queens([ Row/Col | Rest]) :-% otherwise, for each row
       queens(Rest),        % place a queen in each higher numbered
row
       member(Col, [1,2,3,4,5,6,7,8]),     % pick one of the possible
column positions
       safe( Row/Col, Rest).       % and see if that is a safe
       position
                     % if not, fail back and try another column, until
              % the columns are all tried, when fail back to
                     % previous row
safe(Anything, []).  % the empty board is always safe
safe(Row/Col, [Row1/Col1 | Rest]) :-   % see if attack the queen in
next row down
           Col =\= Col1,                % same column?
           Col1 - Col =\= Row1 - Row,  % check diagonal
           Col1 - Col =\= Row - Row1,
           safe(Row/Col, Rest).        % no attack on next row, try
the rest of board
member(X, [X | Tail]).                  % member will pick successive
column values
member(X, [Head | Tail]) :-
           member(X, Tail).
board([1/C1, 2/C2, 3/C3, 4/C4, 5/C5, 6/C6, 7/C7, 8/C8]).     %
prototype board
```

code cite: [16]

In order to solve the N-queen problem in Java, we need to implement some form of backtracking. The following Java code uses for loops and recursion in order to progress through all possible combinations of queens on an 8x8 board. The function gen() keeps track of the current column, and the for statement inside gen() loops through the rows of that column. When a queen can be placed gen() is called on the next column; when the program needs to backtrack it returns to where it was in the for loop of the previous column.

```
class 8-Queen {
	static int n = 8;
	static int[] x = new int[128];
	static boolean[] a = new boolean[128];
	static boolean[] b = new boolean[128];
	static boolean[] c = new boolean[128];

	static void PrintSolution() {
		for (int col=1; col<=n; ++col)
			System.out.print( x[col] );
		System.out.println();
	}
	static void gen( int col ) {
//current column program is in
		for (int row=1; row<=n; ++row)
//loops through rows
			if (a[row] && b[row+col] && c[row-col+n]) {
//if queen can go
//in this row

			x[col] = row; //put queen
			a[row] = b[row+col] = c[row-col+n] = false;
//go to next column, backtracking will return here
			if (col == n) PrintSolution();
			else gen( col+1 );
			a[row] = b[row+col] = c[row-col+n] = true;
			}
	}
	public static void main( String[] args ){
		for (int i=1; i<=2*n; ++i) a[i] = b[i] = c[i] = true;
			gen( 1 );
	}
}
```

LANGUAGES WITH NON-DETEREMINISM

Some examples of languages that support backtracking or “OR” non-determinism are OzLanguage and IconLanguage.

OzLanguage is a multiparadigm programming language that was developed by Smolka in Saarbrücken, Sweden in the early 1990s. This language supports “OR” non-determinism and is used in concurrent constraint programming; German researchers Duchier, Gardent, Niehren discuss methods used to solve combinatorial problems, which vary between two ways: ‘generate and test’ and ‘propagate and distribute’***.

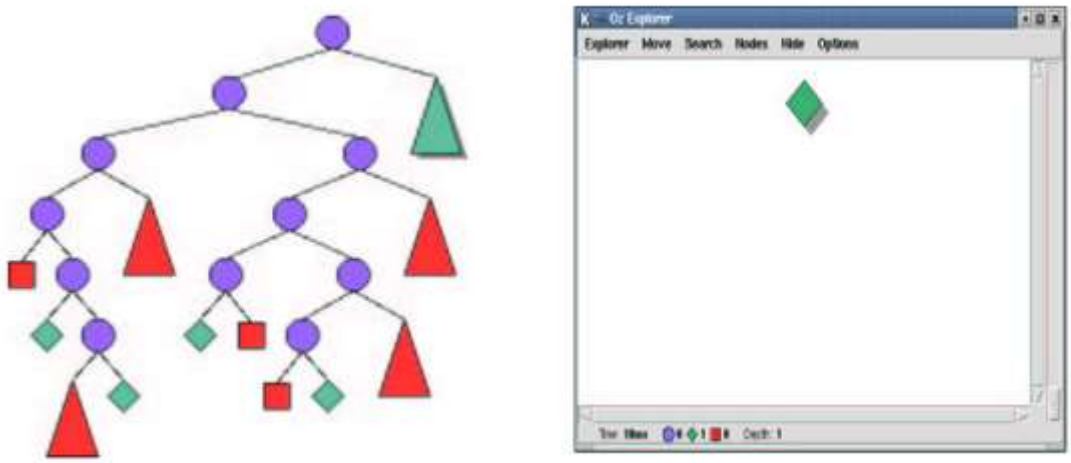


Figure: The tree that results from finding the solutions: Green squares are solutions found to be successful, the red squares are solutions that failed. The purple circles represent choice paths the compiler has to decide between. The left diagram shows the tree generated by 'generate and test' while the right diagram shows the tree generated by 'propagate and distribute'. [13]

For the former case, assignments are enumerated by assigning values to non-assigned variables non-deterministically and is repeated until all possible assignments are found.[13] Time complexity with this method would be exponential as if there are 10 values and there are 10 variables, that would be 10^10 = 10000000000 possible assignments. Although non-determinism assignments were used, it is not efficient to naively use it.

In the latter case, the idea behind propagation is that the set of possible solutions are restricted through deterministic inference while distribution would call on Oz's non-deterministic assignments to evaluate the rest. Unlike 'test and generate', non-determinism was not used naively and thus would achieve a better time complexity by reducing the size of the possible solutions.

Another programming language that has non-determinism or backtracking is Icon. The language was created by Ralph Griswold[14] in 1977 and is multi-paradigm. One feature of Icon are generators, which are the control structures which make up a typical program. As the language puts an emphasis on goal-directed evaluation, generators ensured that they could enumerate values as needed.[15]

4. DATALOG AND RELATION DATABASES

DATALOG

Datalog is:

- a declarative logic language in which each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause.
- a lightweight deductive database system where queries and database updates are expressed in the logic language (PROLOG)[19].

DATALOG VS PROLOG

- Datalog disallows complex terms as arguments of predicates, e.g. p(1,2) is admissible but not p(f(1),2),
- Datalog imposes certain stratification restrictions on the use of negation and recursion (**no recursion involving negation)**[23]
- Datalog requires that every variable that appears in the head of a clause also appears in a non-arithmetic positive literal in the body of the clause.
- Datalog requires that every variable appearing in a negative literal in the body of a clause also appears in some positive literal in the body of the clause.
- Prolog is Turing-complete, whereas Datalog is not.[24]

DATALOG VS SQL

Operation	SQL	DATALOG
Retrieve all tuples	SELECT * FROM People	?- People(name,age)
Restriction	SELECT * FROM People WHERE name='Henry'	?- People(Henry,age)
Restriction + projection	SELECT name FROM People WHERE age = 21	query(name) :- People(name,21) ?- query(name)
Aggregation	SELECT snumber, COUNT(*) AS p_count FROM sp GROUP BY snumber	**Support for mutual recursion precludes internal support for aggregation; aggregate functions would need to be in higher language**

		layer with scoped queries
Insertion	INSERT INTO PEOPLE(name,age) VALUES (‘Wesley’,20), (‘Aahan',21);	People(Wesley,20). People(Aahan,21).
Table creation	CREATE TABLE People (name TEXT NOT NULL PRIMARY KEY, age INTEGER)	**add first entry to 'People' when ready Cannot specify keys.**
Transitive closure (recursion)	**No transitive closure (no recursion)**	path(A,B) :- edge(A,B). path(A,B) :- edge(A,C),path(C,B). ?- path(A,B).
Natural join	SELECT foo,bar FROM red NATURAL JOIN blue;	**NO NATURAL JOIN**
Inner join	SELECT foo, bar FROM red, blue WHERE x = y	query(Foo,Bar) :- red(X,Foo),blue(X,Bar) ?- query(A,B)
Inner join with same-named key	SELECT foo, bar FROM red INNER JOIN blue ON red.x = blue.x	**attributes positional in standard Datalog**

ADVANTAGES OF DATALOG

- Datalog can be used to compute transitive closure, as well as for live variable analysis.[23]
- Because Datalog is not Turing-complete, it is used as a domain-specific language that can take advantage of efficient algorithms developed for query resolution.
- Many important and fundamental problems in database research require rigorous reasoning about queries and their properties. Datalog is much more convenient to use in those cases than SQL or even relational algebra.
  For example, consider an IMDB database. As expected, the database will have dirty data. It is impossible to find a workaround to filter. You can annotate probabilities to the base data, and define rules in Datalog as to how queries should be processed according

to the probability of how accurate they are. Such complex reasoning behind these queries can only be achieved by Datalog.[20]

REFERENCES

[1] A. Roth, "The Practical Application of Prolog," Dr. Dobb's , [Online]. Available: http://www.drdobbs.com/parallel/the-practical-application-of-Prolog/184405220. [Accessed 27 11 2013].

[2] Wikipedia, "Prolog," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Prolog. [Accessed 27 11 2013].

[3] A. Colmerauer, "The Birth of Prolog," [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7438&rep=rep1&type=pdf. [Accessed 27 11 2013].

[4] A. Lally, "Natural Language Processing With Prolog in the IBM Watson System," The Association of Logic Programming, [Online]. Available: http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-Prolog-in-the-ibm-watson-system/. [Accessed 27 11 2013].

[5] J. Eckroth, "Prolog Unificiation," [Online]. Available: http://cse3521.artifice.cc/Prolog-unification.html. [Accessed 27 11 2013].

[6] J. R. Fisher, "Prolog Lists and Sequences," [Online]. Available: http://www.csupomona.edu/~jrfisher/www/Prolog_tutorial/2_7.html. [Accessed 27 11 2013].

[7] D. Byrne, "The Erlang Programming Language," Slideshare, [Online]. Available: http://www.slideshare.net/dennisbyrne/the-erlang-programming-language. [Accessed 27 11 2013].

[8] Bramstein, "JUnify - JavaScript Unification Library," [Online]. Available: http://www.bramstein.com/projects/junify/. [Accessed 27 11 2013].

[9] Fogus, "Unification Versus Pattern Matching... To the Death!," [Online]. Available: http://blog.fogus.me/2010/12/14/unification-versus-pattern-matching-to-the-death/. [Accessed 27 11 2013].

[10] S. Vokes, "Pattern Matching," [Online]. Available: http://c2.com/cgi/wiki?PatternMatching. [Accessed 27 11 2013].

[11] L. McCluskey, "Pattern Matching and Unification," [Online]. Available: http://helios.hud.ac.uk/scomtlm/book/node229.html. [Accessed 27 11 2013].

[12] Roman, "What is 'Pattern Matching' in Functional Languages?," [Online]. Available: http://stackoverflow.com/questions/2502354/what-is-pattern-matching-in-functional-languages. [Accessed 27 11 2013].

[13] D. Duchier, "Concurrent Constraint Programming in Oz for Natural Language Processing," [Online]. Available: http://www.ps.uni-saarland.de/~niehren/index.html/oznlp.pdf. [Accessed 27 11 2013].

[14] R. Perrella, "Icon Language," [Online]. Available: http://c2.com/cgi/wiki?IconLanguage. [Accessed 27 11 2013].

[15] Okmij, "Generators: The API for Traversal and Non-Determinism," [Online]. Available: http://okmij.org/ftp/continuations/generators.html. [Accessed 27 11 2013].

[16] R. Daniels, "NQUEENS - Prolog," [Online]. Available: http://www.cse.scu.edu/~rdaniels/html/courses/Coen171/NQProlog.htm. [Accessed 27 11 2013].

[17] Wikipedia, "Nondeterministic Programming," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Nondeterministic_programming. [Accessed 27 11 2013].

[18] Unknown, "Prolog Programming," [Online]. Available: http://cs.bath.ac.uk/~pb275/CM20019/P/Prolog1.pdf. [Accessed 27 11 2013].

[19] Maier, "Query Language Comparison," [Online]. Available: http://c2.com/cgi/wiki?QueryLanguageComparison. [Accessed 27 11 2013].

[20] Y. K. Hinz, "Datalog Bottom-up is the Trend in the Deductive Database Evaluation Strategy," University of Maryland, [Online]. Available: http://ttnz.cz/statnice/04%20Datalog/other_docs/hinz.pdf. [Accessed 27 11 2013].

[21] S. Ceri, "What You Always Wanted to Know About Datalog," IEEE Transactions on Knowledge and Data Engineering, [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=43410&tag=1. [Accessed 27 11 2013].

[22] MITRE, "Datalog User Manual," MITRE Corporation, [Online]. Available: http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html. [Accessed 27 11 2013].

[23] F. Pfenning, "Logic Programming," [Online]. Available: http://www.cs.cmu.edu/~fp/courses/lp/lectures/26-datalog.pdf. [Accessed 27 11 2013].

[24] Wikipedia, "Datalog," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Datalog. [Accessed 27 11 2013].

[25] I. Bratko, "Introduction to Prolog Programming," [Online]. Available: http://www.cse.unsw.edu.au/~billw/cs9414/notes/Prolog/intro.html. [Accessed 27 11 2013].

[26] H. Ait-Kaci, "Warren's Abstract Machine: A Tutorial Reconstruction," Simon Fraser University, [Online]. Available: http://www.cvc.uab.es/shared/teach/a25002/wambook.pdf. [Accessed 27 11 2013].

[27] Z. Chen, "History on the Implementation and Compilation of Prolog," [Online]. Available: http://www.cs.unm.edu/~pdevineni/papers/Chen.pdf. [Accessed 27 11 2013].

[28] P. Blackburn, "FSAs in Prolog," [Online]. Available: http://cs.union.edu/~striegnk/courses/nlp-with-Prolog/html/node5.html#l1.Prologrepresentation. [Accessed 27 11 2013].