# Some Notes on Popular Logic Programming Languages

**Alireza Dehbozorgi**

https://github.com/alirezadehbozorgi83

https://www.linkedin.com/in/alireza-dehbozorgi-8055702a/

Twitter & Mastadon: 🔗 @BDehbozorgi83

📧 alirezadehbozorgi83@yahoo.com

---

1. **Prolog** (https://en.wikipedia.org/wiki/Prolog) is a widely used logic programming language. It stands for "Programming in Logic". It is a logic-based programming language that is widely used in artificial intelligence, natural language processing, and expert systems. Prolog programs consist of facts, rules, and queries. Facts describe relationships between objects, rules describe how to derive new facts from existing ones, and queries ask questions about the relationships between objects.Here is an example of a simple Prolog program:

```
parent(john, mary).
parent(john, tom).
parent(sue, mary).
parent(tom, ann).

grandparent(X, Y) :-
    parent(X, Z),
    parent(Z, Y).
```

This program defines a few facts about parent-child relationships and a rule to determine grandparent-grandchild relationships.

Below is another simple Prolog code for *List Manipulation*:

```
% Define a predicate to reverse a list
reverse([], []).
reverse([H|T], R) :- reverse(T, TR), append(TR, [H], R).

% Define a predicate to find the last element of a list
last_element([X], X).
last_element([_|T], X) :- last_element(T, X).

% Define a predicate to remove duplicates from a list
remove_duplicates([], []).
remove_duplicates([H|T], R) :- member(H, T),
remove_duplicates(T, R).
remove_duplicates([H|T], [H|R]) :- not(member(H, T)),
remove_duplicates(T, R).
```

Sure, here are some more Prolog snippets:

1. List manipulation

```prolog
% Define a predicate to reverse a list
reverse([], []).
reverse([H|T], R) :- reverse(T, TR), append(TR, [H], R).

% Define a predicate to find the last element of a list
last_element([X], X).
last_element([_|T], X) :- last_element(T, X).

% Define a predicate to remove duplicates from a list
remove_duplicates([], []).

remove_duplicates([H|T], R) :- member(H, T),
remove_duplicates(T, R).
remove_duplicates([H|T], [H|R]) :- not(member(H, T)),
remove_duplicates(T, R).
```

This program defines three predicates for manipulating lists in Prolog. The `reverse` predicate takes a list as input and returns a new list with the elements in reverse order. The `last_element` predicate takes a list as input and returns the last element of the list. The `remove_duplicates` predicate takes a list as input and returns a new list with duplicates removed.

2. Family tree

```
% Define the family relations
male(john).
male(jim).
female(sue).
female(jane).
parent(john, jim).
parent(john, sue).
parent(jim, jane).

% Define the ancestor relation
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This program defines a family tree in Prolog and a predicate for finding ancestors. The `male` and `female` predicates define the genders of the family members. The `parent` predicate defines the parent-child relationships between the family members. The `ancestor` predicate defines the ancestor-descendant relationship between family members. The `ancestor` predicate is defined recursively, so it can find ancestors at any level.

3. Arithmetic operations

```
% Define a predicate to compute the factorial of a number
factorial(0, 1).
factorial(N, F) :- N > 0, N1 is N - 1, factorial(N1, F1), F is F1 * N.

% Define a predicate to compute the Fibonacci sequence
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :- N > 1, N1 is N - 1, fibonacci(N1, F1), N2 is N - 2, fibonacci(N2, F2), F is F1 + F2.
```

This program defines two predicates for performing arithmetic operations in Prolog. The `factorial` predicate takes a number as input and returns its factorial. The `fibonacci` predicate takes a number as input and returns the Nth number in the Fibonacci sequence. Both predicates are defined recursively using the `is` operator to perform arithmetic operations.

Prolog is a powerful and flexiblelanguage for logic programming, and these examples demonstrate some of its capabilities in list manipulation, family tree representation, and arithmetic operations. With its ability to represent and reason about complex logical relationships, Prolog is well-suited for a variety of applications in artificial intelligence, natural language processing, and other areas of computer science.

Another example would be the regarding *Family Tree(s)* :

```
% Define the family relations
male(john).
male(jim).
female(sue).
female(jane).
parent(john, jim).
parent(john, sue).
parent(jim, jane).

% Define the ancestor relation
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This program defines a family tree in Prolog and a predicate for finding ancestors. The `male` and `female` predicates define the genders of the family members. The `parent` predicate defines the parent-child relationships between the family members. The `ancestor` predicate defines the ancestor-descendant relationship between family members. The `ancestor` predicate is

defined recursively, so it can find ancestors at any level.

The 3rd instance is about *Arithmetic Operations* :

```
% Define a predicate to compute the factorial of a number
factorial(0, 1).
factorial(N, F) :- N > 0, N1 is N - 1, factorial(N1, F1), F is F1 * N.

% Define a predicate to compute the Fibonacci sequence
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :- N > 1, N1 is N - 1, fibonacci(N1, F1), N2 is N - 2, fibonacci(N2, F2), F is F1 + F2.
```

This program defines two predicates for performing arithmetic operations in Prolog. The `factorial` predicate takes a number as input and returns its factorial. The `fibonacci` predicate takes a number as input and returns the Nth number in the Fibonacci sequence. Both predicates are defined recursively using the `is` operator to perform arithmetic operations.

Prolog is a powerful and flexiblelanguage for logic programming, and these examples demonstrate some of its capabilities in list manipulation, family tree representation, and arithmetic operations. With its ability to represent and reason about complex logical relationships, Prolog is well-suited for a variety of applications in artificial intelligence, natural language processing, and other areas of computer science.

---

2. **Mercury** (https://en.wikipedia.org/wiki/Mercury_(programming_language) is a logic programming language that is also strongly typed. It is a logic programming language that is also strongly typed. It is designed to be efficient and supports both functional and imperative programming styles. Mercury programs use a syntax similar to Prolog, but with additional features such as type declarations, mode declarations, and determinism annotations. Mercury also supports higher-order programming, module system, and foreign language interfaces. Here is an example of a Mercury program:

```
:- type person ---> john ; mary ; tom.

:- pred loves(person::in, person::in).
:- mode loves(in, in) is semidet.

loves(john, mary).
loves(mary, john).
loves(tom, mary).
```

This program defines a type of person and a predicate `loves` to determine if one person loves another. Here are some more Mercury snippets:

a. List manipulation

```
% Define a predicate to reverse a list
:- pred reverse(list(T), list(T)).
:- mode reverse(in, out) is det.
reverse([], []).
reverse([H|T], R) :- reverse(T, TR), R = TR ++ [H].

% Define a predicate to find the last element of a list
:- pred last_element(list(T), T).
:- mode last_element(in, out) is semidet.
last_element([X], X).
last_element([_|T], X) :- last_element(T, X).
```

```
12
13  % Define a predicate to remove duplicates from a list
14  :- pred remove_duplicates(list(T), list(T)).
15  :- mode remove_duplicates(in, out) is det.
16  remove_duplicates([], []).
17  remove_duplicates([H|T], R) :- member(H, T),
    remove_duplicates(T, R).
18  remove_duplicates([H|T], [H|R]) :- not member(H, T),
    remove_duplicates(T, R).
```

This program defines three predicates for manipulating lists in Mercury. The `reverse` predicate takes a list as input and returns a new list with the elements in reverse order. The `last_element` predicate takes a list as input and returns the last element of the list. The `remove_duplicates` predicate takes a list as input and returns a new list with duplicates removed.

b. Merge sort

```
1   % Define a predicate to perform merge sort on alist
2   :- pred merge_sort(list(T), list(T)).
3   :- mode merge_sort(in, out) is det.
4   merge_sort([], []).
5   merge_sort([X], [X]).
6   merge_sort(List, Sorted) :-
7       List = [_, _|_],
8       split(List, Left, Right),
9       merge_sort(Left, LeftSorted),
10      merge_sort(Right, RightSorted),
11      merge(LeftSorted, RightSorted, Sorted).
12
13  % Define a predicate to split a list into two halves
14  :- pred split(list(T), list(T), list(T)).
15  :- mode split(in, out, out) is det.
```

```
split([], [], []).
split([X], [X], []).
split([X, Y|T], [X|LT], [Y|RT]) :- split(T, LT, RT).

% Define a predicate to merge two sorted lists
:- pred merge(list(T), list(T), list(T)).
:- mode merge(in, in, out) is det.
merge([], Right, Right).
merge(Left, [], Left).
merge([L|LT], [R|RT], [L|Sorted]) :- L =< R, merge(LT,
[R|RT], Sorted).
merge([L|LT], [R|RT], [R|Sorted]) :- L > R, merge([L|LT],
RT, Sorted).
```

This program defines a predicate for performing merge sort on a list in Mercury. The `merge_sort` predicate takes alist as input and returns a new list with the elements sorted in ascending order. The `split` predicate takes a list as input and splits it into two halves. The `merge` predicate takes two sorted lists as input and merges them into one sorted list.

c. Prime numbers

```
% Define a predicate to check if a number is prime
:- pred is_prime(int::in) is semidet.
is_prime(N) :-
    N > 1,
    not has_factor(N, 2).

% Define a predicate to check if a number has a factor
:- pred has_factor(int::in, int::in) is semidet.
has_factor(N, F) :-
    F * F =< N,
    ( N mod F = 0 ; has_factor(N, F + 1) ).
```

This program defines two predicates for generating prime numbers in Mercury. The `is_prime` predicate takes a number as input and returns true if the number is prime. The `has_factor` predicate takes a number and a factor as input and returns true if the number has the factor. These predicates use a recursive approach to check if the number is prime by checking if it has any factors.

Mercury is a logic programming language with a strong type system and a declarative syntax. These examples demonstrate some of its capabilities in list manipulation, sorting, and prime number generation. With its combination of logic and functional programming, Mercury is well-suited for a varietyof applications in software engineering, scientific computing, and other areas of computer science.

---

3. **Datalog** (https://en.wikipedia.org/wiki/Datalog) is a declarative logic programming language that is used mainly for database queries. It is a declarative logic programming language that is used mainly for database queries. It is a subset of Prolog and is based on the idea of deductive databases. Datalog programs consist of facts and rules, and queries are written in the form of patterns that match against the facts and rules. Datalog is often used in data analytics, business intelligence, and knowledge management applications. Here is an example of a Datalog program:

```
likes(john, mary).
likes(tom, mary).
likes(jane, tom).

friend(X, Y) :- likes(X, Z), likes(Y, Z).
```

This program defines a few facts about who likes whom and a rule to determine if two people are friends based on a common liking. Here are some more examples of Datalog programs:

a. Family relationships

```
parent(john, jim).
parent(john, sue).
parent(jim, jane).
parent(sue, ann).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This Datalog program defines a set of family relationships using two predicates: `parent` and `ancestor`. The `parent` predicate describes the parent-child relationship between individuals, and the `ancestor` predicate defines the ancestor-descendant relationship. The `ancestor` predicate is defined recursively using the parent-child relationship.

b. Graph traversal

```
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(e, f).

reach(X, Y) :- edge(X, Y).
reach(X, Y) :- edge(X, Z), reach(Z, Y).
```

This Datalog program defines a directed graph using the `edge` predicate and a reachability predicate called `reach`. The `reach` predicate is used to determine if there is a path from node `X` to node `Y` in the graph. It is defined recursively using the `edge` predicate.

c. Student grades

```
grade(john, math, 95).
grade(john, science, 85).
grade(sue, math, 80).
grade(sue, science, 90).

passingSorry about that, it looks like my response got cut off. Here are the complete snippets for the student grades example:
```

```
grade(john, math, 95).
grade(john, science, 85).
grade(sue, math, 80).
grade(sue, science, 90).

passing(X, Y) :- grade(X, Y, Z), Z ≥ 70.
```

This Datalog program defines a set of student grades using the `grade` predicate. The `passing` predicate is used to determine if a student has passed a particular course, based on their grade. It is defined using the `grade` predicate and a condition that the grade must be greater than or equal to 70.

Datalog is a declarative programming language used for database queries and knowledge representation. These examples demonstrate some of its capabilities in defining family relationships, graph traversal, and student grades. With its emphasis on logical relationships and rule-based reasoning, Datalog is well-suited for a variety of applications in artificial intelligence, data analytics, and other fields.

---

1. **Alloy** (https://en.wikipedia.org/wiki/Alloy_(specification_language) is a declarative language for modeling and analyzing software systems. It is a declarative language for modeling and analyzing software systems. It is based on the idea of relational logic and is used to specify constraints and properties of software systems. Alloy programs consist of signatures, facts, and constraints. Signatures define sets of objects, facts describe relationships between objects, and constraints specify properties that must hold for the objects. Alloy also includes a graphical user interface for visualizing and exploring models, and a model checker for verifying properties of models. Alloy is often used in software engineering research and education for modeling and analyzing software architectures, designs, and requirements. Here is an example of an Alloy program:

```
sig Person {
  friends: set Person
}
fact {
  all p: Person | p not in p.friends
}
pred mutualFriends[p, q: Person] {
  p.friends & q.friends != none
}
run mutualFriends for 5
```

This program defines a Person signature with a set of friends, a fact to ensure that a person is not their own friend, and a predicate to find mutual friends between two people. The `run` command is used to execute the `mutualFriends` predicate with a specified number of instances. Below are some more examples:

1. *Model for a simple social network*

```
sig Person {
  friends: set Person
}

pred showFriends(p: Person) {
  #p.friends ≥ 2
}

run showFriends for 5
```

This Alloy model defines a `Person` signature with a `friends` relation that represents the set of friends for each person. The `showFriends` predicate is used to find all people who have at least two friends, and the `run` command executes the predicate for a universe of size 5.

2. *Model for a simple bank account system*

```
sig Account {
  balance: Int
}

pred withdraw[a: Account, amount: Int] {
  amount > 0 and amount ≤ a.balance
  a.balance = a.balance - amount
}

run withdraw for 5 but exactly 2 Account
```

This Alloy model defines an `Account` signature with an `Int` attribute `balance` that represents the account balance. The `withdraw` predicate is used to simulate a withdrawal from an account, and the `run` command executes the predicate for a universe of size 5 with exactly 2 accounts.

3. *Model for a simple file system*

```
sig File { }
sig Directory {
  contents: set File + set Directory
}

pred showRoot(d: Directory) {
  no d.^contents & Directory
}
```

run showRoot for 5 but exactly 1 FileThis Alloy model defines a `File` signature and a `Directory` signature with a `contents` relation that represents the set of files and directories inside the directory. The `showRoot` predicate is used to find all directories that have no subdirectories as their direct

or indirect contents, and the `run` command executes the predicate for a universe of size 5 with exactly 1 file.

Alloy is a formal modeling language that is used for software engineering and other applications that require formal specification and verification of systems. These examples demonstrate some of its capabilities in modeling social networks, bank account systems, and file systems. With its emphasis on precise specification and logical reasoning, Alloy is well-suited for a variety of applications in software engineering, computer science, and other fields.

---

5. **ASP (Answer Set Programming)** (https://en.wikipedia.org/wiki/Answer_set_programming) ASP is a declarative programming language that is used for solving combinatorial optimization problems. ASP programs consist of rules and constraints that specify a problem instance. The ASP solver then generates "answer sets" that satisfy the rules and constraints. ASP is often used in artificial intelligence and operations research for tasks such as planning, scheduling, and decision making. Here's an example of an ASP program that solves a simple optimization problem:

```
% Define the input data
n(3).
m(2).
p(1..n).
q(1..m).

% Define the decision variables
1 { x(P,Q) : q(Q) } 1 :- p(P).
1 { y(Q) : p(P), x(P,Q)=1 } 1 :- q(Q).

% Define the objective function
minimize sum{ x(P,Q) : p(P), q(Q) }.

% Define the constraints
```

```
:- x(P,Q), x(P,Q'), Q<Q'.
:- x(P,Q), x(P',Q), P<P'.
```

This program defines a simple optimization problem where we want to assign each of `n` tasks to one of `m` machines. The decision variables `x(P,Q)` represent whether task `P` is assigned to machine `Q`. The variable `y(Q)` is a helper variable used to ensure that each machine is assigned exactly one task. The objective function is to minimize the total number of tasks assigned to machines. The constraints ensure that each task is assigned to exactly one machine, and that no two tasks are assigned to the same machine.

To solve this problem, we would save the code above in a file with a `.lp` extension, and then use an ASP solver such as Clingo to generate the optimal solution. The solver would find an assignment of tasks tomachines that satisfies the constraints and minimizes the objective function.

---

5. **ACL2 (A Computational Logic for Applicative Common Lisp)** (https://en.wikipedia.org/wiki/ACL2) ACL2 is a programming language and theorem prover that is used for verifying the correctness of computer systems. ACL2 programs consist of functions and specifications that describe the behavior of the system. The ACL2 theorem prover then checks that the specifications are met by the functions. ACL2 is often used in computer science research and education for teaching formal methods and verifying software and hardware systems. Here's an example of an ACL2 program:

```
;; Define a function to compute the factorial of a number
(defun factorial (n)
  (if (zerop n)
      1
      (* n (factorial (1- n)))))

```

```
;; Define a theorem to prove that the factorial function is correct
(defthm factorial-correctness
  (implies (and (integerp n) (≥ n 0))
           (= (factorial n) (apply '* (range 1 (1+ n))))))

;; Define a function to compute the nth Fibonacci number
(defun fibonacci (n)
  (cond ((zerop n) 0)
        ((= n 1) 1)
        (t (+ (fibonacci (- n 1)) (fibonacci (- n 2))))))

;; Define a theorem to prove that the Fibonacci function is correct
(defthm fibonacci-correctness
  (implies (integerp n)
           (= (fibonacci n)
              (cond ((zerop n) 0)
                    ((= n 1) 1)
                    (t (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))))
```

This program defines two functions, `factorial` and `fibonacci`, and two theorems, `factorial-correctness` and `fibonacci-correctness`, toprove the correctness of these functions.

The `factorial` function computes the factorial of a number `n` by recursively multiplying `n` with the factorial of `n-1`. The `factorial-correctness` theorem proves that the factorial function is correct for non-negative integer inputs by showing that the function's result is equal to the product of all integers from 1 to `n`.

The `fibonacci` function computes the nth Fibonacci number by recursively summing the (n-1)th and (n-2)th Fibonacci numbers. The `fibonacci-correctness` theorem proves that the Fibonacci function is correct for all integer inputs by induction on `n`, showing that the function's result is equal to the sum of the (n-1)th and (n-2)th Fibonacci numbers.

ACL2 is a programming language and theorem prover that is used for verifying the correctness of computer systems. It is based on Common Lisp and uses a combination of functional programming, logic programming, and theorem proving techniques to model and reason about computer programs. ACL2 is often used in computer science research and education for teaching formal methods and verifying software and hardware systems.

---

7. **CHR (Constraint Handling Rules)** (https://en.wikipedia.org/wiki/Constraint_Handling_Rules#:~:text=A%20CHR%20program%2C%20sometimes%20called,the%20state%20of%20the%20program.) CHR is a declarative programming language that is used for solving constraint satisfaction problems. CHR programs consist of rules that describe how to simplify constraints and propagate information. CHR is often used in artificial intelligence and operations research for tasks such as scheduling, planning, and resource allocation. Here's an example of a Constraint Handling Rules (CHR) program:

```
:- chr_constraint path/3, shortest/2.
:- op(700, xfy, --->).

path(X, Y, L) ---> edge(X, Y, L).
path(X, Z, L) ---> edge(X, Y, L1), path(Y, Z, L2), { L = L1 + L2 }.

shortest(X, L1), shortest(X, L2) <=> L1 =< L2 | true.
shortest(X, L) \ shortest(Y, L2) ==> path(X, Y, L1), { L1 < L2 } | shortest(Y, L1).
```

This program defines a simple graph search algorithm to find the shortest path between two nodes in a weighted directed graph. The `path/3` constraint represents a path between two nodes with a certain length, and the `shortest/2` constraint represents the current shortest path length from a given node. The program uses the following rules:

- If there is a direct edge between two nodes, then there is a path of length equal to the weight of the edge.
- If there is a path from `X` to `Y` with length `L1`, and there is a path from `Y` to `Z` with length `L2`, then there is a path from `X` to `Z` with length `L1 +L2`.

The `shortest/2` constraints are used to keep track of the current shortest path lengths from a given node. The first rule ensures that if there are two `shortest/2` constraints for the same node, then the one with the smaller length is kept. The second rule ensures that if there is a shorter path from `X` to `Y` than the current shortest path from `Y`, then the shortest path length from `Y` is updated to the length of the new path.

To use this program, we would add the constraints `path(X, Y, L)` and `shortest(X, L)` to the system, where `X` and `Y` are the starting and ending nodes, and `L` is the length of the shortest path. We would then use a CHR solver to generate the optimal solution. The solver would find the shortest path between

the two nodes by iteratively adding new constraints and applying the rules until a solution is found.

---

5. **Oz** (https://en.wikipedia.org/wiki/Oz_(programming_language)) Oz is a programming language that combines logic programming, functional programming, and concurrent programming. Oz programs consist of constraints, functions, and processes that communicate via message passing. Oz is often used in computer science research and education for teaching concurrency, distributed systems, and programming language paradigms. Here's an example of an Oz program that demonstrates some of the features of the language:

```
functor
import
   Application,
   Browser,
   System,
   Thread,
   Xlib
   at 'lib/System'
   at 'lib/Thread'
define
   % Define a function to draw a square using Xlib
   fun {DrawSquare X Y Size}
      Xlib.drawLine(X, Y, X + Size, Y)  % Top
      Xlib.drawLine(X + Size, Y, X + Size, Y + Size)  % Right
      Xlib.drawLine(X + Size, Y + Size, X, Y + Size)  % Bottom
      Xlib.drawLine(X, Y + Size, X, Y)  % Left
   end

   % Define a function to generate a list of random numbers
```

```
   fun {RandomList N Max}
      List = []
      repeat N
         List = List ++ {System.rand(Max)}
      end
      List
   end

   % Define a function to sort a list of numbers using quicksort
   fun {Quicksort L}
      case L
      of [] then []
      [] [X | Xs] then
         L1, L2 = {Partition X Xs}
         {Quicksort L1} ++ [X] ++ {Quicksort L2}
      end
   end

   % Define a function to partition alist of numbers around a pivot value for use in quicksort
   fun {Partition Pivot L}
      L1 = [X || X in L | X < Pivot]
      L2 = [X || X in L | X > Pivot]
      L1, L2
   end

   % Generate a random list of numbers and sort it using quicksort
   L = {RandomList 10 100}
   {Browser.showInfoDialog L}
   Sorted = {Quicksort L}
   {Browser.showInfoDialog Sorted}

   % Draw a square using Xlib
   {DrawSquare 50 50 50}

```

```
    % Create a thread to perform a long-running computation
    Thread.fork
       proc {LongComputation}
          {System.sleep 10000}
          {Browser.showInfoDialog 'Computation finished!'}
       end
    end

    % Start the Oz application loop
    {Application.run}
 end
```

This program demonstrates several features of the Oz programming language, including:

- Importing libraries and modules using the `import` statement
- Defining functions using the `fun` keyword
- Using pattern matching to handle different cases in a function definition
- Generating random numbers using the `System.rand` function
- Sorting a list of numbers using the quicksort algorithm
- Drawing a square using the Xlib library
- Creating a thread to perform a long-running computation using the `Thread.fork` statement
- Using the `System.sleep` function to pause execution for a certain amount of time
- Displaying information dialogs using the `Browser.showInfoDialog` function
- Starting the Oz application loop using the `Application.run` function

When executed, this program generates a list of 10 random numbers between 0 and 100, sorts them using quicksort, displays the original and sorted lists in information dialogs, draws a square using Xlib, and creates a thread to perform a long-running computation that displays an information dialog when finished. The Oz application loop is then started to handle user interaction and events.

---

5. **F-Logic (Frame-Based Logic)** (https://en.wikipedia.org/wiki/F-logic) F-Logic is a logic programming language that is used for knowledge representation and reasoning. F-Logic programs consist of frames, which describe objects and their properties, and rules, which describe relationships between objects. F-Logic is often used in artificial intelligence and knowledge management for tasks such as ontology modeling, semantic web, and natural language processing. Here are some examples of F-Logic programs:

1. *Family relationships*

```
class person {
  isa entity
}

class male {
  isa person
}

class female {
  isa person
}

class parent {
  isa relation
  inverse child
  plays father, mother
}
```

```
18
19  class child {
20    isa relation
21    plays son, daughter
22  }
23
24  male(john).
25  female(sue).
26  male(jim).
27  female(jane).
28  parent(john, jim).
29  parent(john, sue).
30  parent(jim, jane).
31  parent(sue, ann).
32
33  ancestor(X, Y) :- parent(X, Y).
34  ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This F-Logic program defines a set of family relationships using classes and relations. The `person` class is used to represent individuals, and the `male` and `female` classes are used to represent genders. The `parent` and `child` classes are used to represent the parent-child relationship.

2. *Graph traversal*

```
1  class node {
2    isa entity
3  }
4
5  class edge {
6    isa relation
7    inverse edge
8    plays from, to
```

```
}

node(a).
node(b).
node(c).
node(d).
node(e).
node(f).

edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(e, f).

reachable(X, Y) :- edge(X, Y).
reachable(X, Y) :- edge(X, Z), reachable(Z, Y).
```

This F-Logic program defines a directed graph using the `node` and `edge` classes. The `reachable` rule is used to determine if there is a path from node `X` to node `Y` in the graph. It is defined recursively using the `edge` relation.

3. *Student grades*

```
class student {
  isa entity
}

class course {
  isa entity
}

class grade {
  isa relation
```

```
    inverse grade_of
    plays student, course
    grade_value: integer
  }

  student(john).
  student(sue).
  course(math).
  course(science).

  grade(john, math, 95).
  grade(john, science, 85).
  grade(sue, math, 80).
  grade(sue, science, 90).

  passing(X, Y) :- grade(X, Y, GV), GV ⩾ 70.
```

This F-Logic program defines a set of student grades using classes and relations. The `student` and `course` classes are used to represent the students and courses, respectively. The `grade` relation is used to represent the grades for each student in each course.

The `passing` rule is used to determine if a student has passed a particular course, based on their grade. It is defined using the `grade` relation and a condition that the grade must be greater than or equal to 70.

### F-Logic and Natural Language Processing (NLP/NLProc)

F-Logic can be used in natural language processing (NLP) to represent knowledge about language and to perform reasoning on that knowledge. Here's an example of how F-Logic can be used to represent knowledge about the grammatical structure of sentences:

```
  class word {
```

```
  isa entity
  text: string
}

class noun {
  isa word
}

class verb {
  isa word
}

class subject {
  isa entity
  plays subject_of
}

class object {
  isa entity
  plays object_of
}

class sentence {
  isa entity
  has_subject: subject
  has_verb: verb
  has_object: object
}

word(the, "the").
word(cat, "cat").
word(sat, "sat").
word(on, "on").
word(the, "the").
word(mat, "mat").

noun(cat).
```

```
noun(mat).

verb(sat).

subject(cat).
object(mat).

sentence(S) :- S.has_subject = cat, S.has_verb = sat,
S.has_object = mat.
```

This F-Logic program defines a set of classes for words, nouns, and verbs, as well as classes for subjects, objects, and sentences. The `word` class has an attribute `text` that represents the text of the word. The `noun` and `verb` classes are used to represent nouns and verbs, respectively. The `subject` and `object` classes are used to represent the subject and object ofa sentence, respectively. The `sentence` class has attributes `has_subject`, `has_verb`, and `has_object` which are used to represent the subject, verb, and object of a sentence, respectively.

The last line of the program defines a rule that matches a sentence where the subject is `cat`, the verb is `sat`, and the object is `mat`.

With this knowledge representation, we can perform reasoning on sentences to identify their grammatical structure and to extract meaning from them. For example, we can use this F-Logic program to parse the sentence "The cat sat on the mat" and to infer that the subject is "cat", the verb is "sat", and the object is "mat". We can also use this knowledge to generate sentences that follow the same grammatical structure.

F-Logic provides a powerful way to represent and reason about knowledge in a variety of domains, including natural language processing. Its object-oriented syntax and emphasis on logical reasoning make it well-suited for a variety of applications in artificial intelligence, knowledge representation, and other fields.

---

5. **λProlog (Lambda Prolog)** (https://en.wikipedia.org/wiki/%CE%9BProlog) (https://www.site.uottawa.ca/~afelty/dist/lprolog97.pdf) λProlog is a programming language that combines logic programming and functional programming. λProlog programs consist of logic formulas and lambda expressions that can be used as functions. λProlog is often used in computer science research and education for teaching programming language semantics and type systems. Here's a simple example of a Lambda Prolog program:

```
% Define a function to compute the factorial of a number
pred factorial(n : nat, f : nat → nat)
   fact : nat = 1
   mult : nat → nat → nat = \x y → x * y
   f : nat → nat = \x → if x = 0 then fact else mult x (f (x - 1))
end

% Define a function to compute the nth Fibonacci number
pred fibonacci(n : nat, f : nat → nat)
   fib : nat = 0
   next : nat → nat → nat = \x y → x + y
   f : nat → nat = \x →
      if x = 0 then fib
      else if x = 1 then 1
      else next (f (x - 1)) (f (x - 2))
end

% Define a theorem to prove that the factorial function is correct
thm factorial-correctness : all n : nat . (factorial n (\x : nat . x)) n = product (range 1 n)

% Define a theorem to prove that the Fibonacci function is correct
```

```
22    thm fibonacci-correctness : all n : nat . (fibonacci n (\x
      : nat . x)) n = fib n
```

This program defines two functions, `factorial` and `fibonacci`, and two theorems, `factorial-correctness` and `fibonacci-correctness`, to prove the correctness of these functions.

The `factorial` function computes the factorial of a number `n` using a recursive lambda expression. It uses an auxiliary function `mult` to compute the product of two numbers. The `factorial-correctness` theorem proves that the factorial function is correct for all non-negative integer inputs by showing that the function's result is equal to the product of all integers from 1 to `n`.

The `fibonacci` function computes the nth Fibonacci number using a recursive lambda expression. It uses an auxiliary function `next` to compute the sum of two numbers. The `fibonacci-correctness` theorem proves that the Fibonacci function is correct for all integer inputs by induction on `n`, showing that the function's result is equal to the nth Fibonacci number.

Lambda Prolog is a logical programming language that combines the features of lambda calculus and logic programming. It is used for formal verification and theorem proving, as well as for programming tasks that require higher-order functions and recursion. The language provides a rich set of features for defining functions and predicates, working with logical formulas, and proving theorems.

---

1. **MiniZinc** (https://www.minizinc.org/) MiniZinc is a constraint programming language that is used to model and solve combinatorial optimization problems. MiniZinc programs consist of constraints and variables that describe the problem instance. The MiniZinc solver then generates solutions that satisfy the constraints. MiniZinc is often used in operations research, scheduling, and planning. Here's a simple example of a MiniZinc program:

```
% Define the size of the Sudoku grid
int: n = 9;

% Define the decision variables
array[1..n, 1..n] of var 1..n: grid;

% Define the constraints for rows, columns, and sub-grids
constraint forall(i in 1..n) (all_different([grid[i,j] | j in 1..n]));
constraint forall(j in 1..n) (all_different([grid[i,j] | i in 1..n]));
constraint forall(k in 0..2, l in 0..2) (all_different([grid[i,j] | i in 1+3*k..3*(k+1), j in 1+3*l..3*(l+1)]));

% Define the input data
% This is a sample Sudoku puzzle, with 0s representing empty cells
array[1..n, 1..n] of int: input =
[
    0, 0, 0, 2, 0, 0, 0, 6, 3,
    3, 0, 0, 0, 0, 5, 4, 0, 1,
    0, 0, 1, 0, 0, 3, 9, 8, 0,
   % ... continued from previous example ...

    0, 0, 0, 0, 0, 0, 0, 9, 0,
    0, 0, 0, 5, 3, 8, 0, 0, 0,
    0, 3, 0, 0, 0, 0, 0, 0, 0,
    0, 2, 6, 3, 0, 0, 5, 0, 0,
    5, 0, 3, 7, 0, 0, 0, 0, 8,
    4, 7, 0, 0, 0, 1, 0, 0, 0
];

% Apply the input data as constraints
constraint forall(i in 1..n, j in 1..n where input[i,j] > 0) (grid[i,j] = input[i,j]);
```

```
31
32    % Solve the Sudoku puzzle
33    solve satisfy;
34
35    % Output the solution
36    output [show(grid)];
```

This program defines a Sudoku puzzle using MiniZinc and solves it using the `satisfy` solver. The program uses the following features of MiniZinc:

- Defining decision variables using the `var` keyword
- Specifying the domains of decision variables using `int` and `array` declarations
- Defining constraints using `constraint forall` and `all_different`
- Specifying input data using an `array` declaration with integer values
- Applying input data as constraints using a `constraint forall` loop with a `where` condition
- Solving the puzzle using the `solve` statement with the `satisfy` solver
- Outputting the solution using the `output` statement with the `show` function

When executed, this program solves a Sudoku puzzle specified by the `input` array, which represents the initial configuration of the puzzle with 0s representing empty cells. The solver applies the row, column, and sub-grid constraints to find a valid solution that satisfies all the constraints. The solution is then output using the `show` function, which displays the values of the `grid` array in a readable format.

Note that this program is just a simple example, and there are many more advanced Sudoku puzzles and solving techniques that can be implemented using MiniZinc.

---

1. **IDP (Intensional Declarative Programming)** (https://docs.idp-z3.be/en/0.8.3/introduction.html)( https://arxiv.org/abs/1511.00916) IDP is a declarative programming language that is used for knowledge representation and reasoning. IDP programs consist of logic formulas and modal operators that describe the properties of objects and their relationships. IDP is often used in artificial intelligence, knowledge management, and database systems. It stands for "Integrating Deductive and Probabilistic reasoning". It is a knowledge representation and reasoning system developed at the KU Leuven in Belgium. IDP is designed to integrate different types of reasoning, including deductive reasoning (based on logic and rules) and probabilistic reasoning (based on probabilities and statistics).

   IDP supports various logic-based formalisms, including first-order logic, modal logic, and answer set programming. It provides a unified language for specifying theories and structures in these formalisms, and it can automatically generate models and proofs for these specifications. IDP also provides tools for visualization, debugging, and optimization of models.

   IDP has been used for a wide range of applications, including knowledge representation, planning, diagnosis, and decision making. It has been applied in various domains, such as robotics, healthcare, and finance. IDP is also used as a teaching tool for logic and knowledge representation.

   One of the unique features of IDP is its ability to integrate different types of reasoning. For example, IDP can combine logical rules and probabilistic models to reason about uncertain domains. This makes IDP suitable for applications that require both deductive and probabilistic reasoning, such as decision making under uncertainty.

   Overall, IDP is a powerful and flexible system for knowledge representation and reasoning. Its ability to integrate different types of reasoning makes it a valuable tool for various applications in AI and other fields.

2. Here's a simple example of an IDP program:

```
vocabulary V {
  type symbol
  constant a, b, c : symbol
}

theory T : V {
  a ≠ b
  b ≠ c
}

structure S : V {
  a → 1
  b → 2
  c → 3
}

procedure main()
  models(T, S)
end
```

This program defines a simple theory and structure using the IDP language. The program uses the following features of IDP:

- Defining a vocabulary using the `vocabulary` statement
- Defining symbols and constants using the `type` and `constant` keywords
- Defining a theory using the `theory` statement
- Specifying constraints using the `≠` operator
- Defining a structure using the `structure` statement
- Specifying interpretations for symbols using the `→` operator
- Calling the `models` procedure to generate models for the theory and structure

When executed, this program generates all possible models for the theory and structure, which satisfy the constraints specified in the theory. In this case, there are only two possible models:

```
{a → 1, b → 2, c → 3}
{a → 1, b → 3, c → 2}
```

These models satisfy the constraints that `a` is not equal to `b` and `b` is not equal to `c`. The first model assigns `a` to 1, `b` to 2, and `c` to 3, while the second model assigns `a` to 1, `b` to 3, and `c` to 2.

IDP is a knowledge representation and reasoning system that supports various logic-based formalisms, including first-order logic, modal logic, and answer set programming. It can be used for a wide range of applications, including knowledge representation, planning, diagnosis, and decision making.

---

Note that each of these logic programming languages has its own strengths and weaknesses, and is suited to different types of programming tasks.

**To be continued** ....

---