Introduction:

Logic programming is a programming paradigm that's employed by most of the computer scientists to provide the machines with the ability to reason. Logic programming was developed by a professor at the University of Edinburgh in 1974. Logic programming will be interfaced with different programming languages like C or Java. Logic is a language that has semantics and syntax. The syntax is nothing but the rules that tell us regarding a way to form formulas. Semantics tells us regarding the meaning conveyed by the formulas. Logic has inference rules. Inference rules describe the right ways of concluding. Logic is employed to represent knowledge. Knowledge is manipulated by using a resolution inference system. Resolution may be a primary rule of inference in logic programming. The resolution will be slow on an outsized database. Resolution reasoning is employed to prove theorems in the clausal form.

The logic is typically in the clausal form which is a subset of first-order predicate logic. The first order predicate logic is simple to grasp and be employed to represent all the computer issues. The languages utilized in logic programming also are referred to as declarative languages. A logic program is an assortment of Horn clauses. Execution of the program logic may be a method of proving theorems and computation is done using logic inferences. Computation in a logic program consists of testing a given assertion (query) A. The testing of a query A will be enforced by a way referred to as resolution. The method of determining helpful values for variables is named unification. Resolution uses unification. Unification involves backtracking. The resolution needs a temporary assignment of values to variables, and also the method is named representation. For instance:

mammal(human) ⊂ {}.

{} ⊂ mammal(human).

mammal(human) ⊂ mammal(human).

{}⊂ {}.

In Logic programming, facts and rules are used to express problems. Logic programming languages also are referred to as declarative languages. Prolog is a declarative language. Logic programming finds its applications in logical reasoning, problem-solving, crypt-arithmetic issues, GRE analytical issues, equation logic programming languages (e.g. OBJ3 or Equation Interpreter Project), solving puzzles, within the field of computing, specification verification (or speedy prototyping), etc. Logic programs don't specify the execution sequence as in most in the case of other language paradigms.

Solving problems with Logic Programming:

Facts are true statements regarding the program and also the data. For instance, Paris is the capital of France. Facts and rules are used for solving problems. Facts are expressed as logical clauses and have a head solely. Rules are constraints that aid in concluding about the problem domain. Rules are specified as logical clauses with a head and a body. They're used for expressing facts. For instance, ancestor(A, B):- descendent(X, B). The syntax for rules is G: X1, X2,..Xn. Facts and rules are used for solving problems. Logic programs don't specify the execution sequence like other language paradigms. The user specifies the specifications of a solution and the computer derives the execution sequence for that solution. A logic program may be an assortment of horn clauses. A Horn clause may be a preposition with zero or one terms in the consequent.

About Kanren Package:

It gives us an approach to streamlining the manner in which we made the code for business rationale. The accompanying direction will enable you to introduce Kanren. Kanren is an inserted Domain Specific Language for rationale programming. The center Kanren language is straightforward with just three coherent administrators and one interface administrator. The center language, utilizing Scheme as the host language, is depicted in this short, intelligent instructional exercise. Kanren has been executed in an exceedingly developing number of host dialects, as well as a scheme, Racket, Clojure, Haskell, Python, JavaScript, Scala, Ruby, OCaml, and PHP, among numerous totally different dialects. Kanren is intended as effectively adjusted and broadened; augmentations incorporate Constraint Logic Programming, probabilistic rationale programming, ostensible rationale programming, and postponing. Kanren was structured as a negligible rationale programming language, with a little, effectively justifiable, and effectively hackable usage.

Kanren was at first embedded in the scheme and has been ported to numerous other host tongues over the earlier decade. The most outstanding Kanren execution is 'core.logic' in Clojure, which presently has numerous Prolog-like expansions and various improvements. As of late the center of the Kanren execution has been rearranged significantly further, bringing about a little “small scale piece” called “microKanren.” Kanren would then be able to be actualized over this microKanren center. Therefore, most mainstream abnormal state dialects have at any rate one Kanren or microKanren usage.

The standard executions of Kanren and microKanren contain no change or opposite symptoms, with a solitary exemption: a few forms of Kanren use pointer balance for correlation of rational factors. I look at this as a “considerate impact,” albeit numerous executions stay away from even this impacted by going a counter through the usage. There is likewise no worldwide reality database. Kanren's execution theory is propelled by utilitarian programming: transformation and impacts ought to be stayed away from, and all language builds should regard lexical extension. On the off chance that you take a gander at the usage, you may even detect two or three monads. The inquiry usage depends on consolidating and controlling apathetic streams, indeed without utilizing transformation. These usage decisions lead to an altogether different exchange offs than in Prolog. In Prolog, the variable query is a steady time, yet backtracking requires fixing reactions. In Kanren variable query is progressively costly, however, backtracking is “free.” Truth be told, there is no backtracking in Kanren, because of how the streams are taken care of.

One intriguing part of the Kanren execution is the code naturally strings safe and from a certain perspective inconsequentially parallelizable. Obviously, parallelizing the code without making it slower isn't unimportant, given that each string or procedure must be given enough work to compensate for the overhead of parallelization. All things considered, this is a zone of Kanren execution that I expect will get more consideration and experimentation.

Kanren utilizes the happen check for unification, and utilization a total interleaving look rather than profundity first hunt. Interleaving pursuit utilizes more memory than profundity first hunt, however, can discover answers now and again in which profundity first inquiry will veer/circle until the end of time. Kanren supports a couple of additional intelligent administrators like conda, condu, and venture, for instance. Conda and condu can be utilized to recreate Prolog's cut, and the task can be utilized to get the esteem related with a rationale variable.

The nearness of conda, condu, and venture and the capacity to effectively adjust the inquiry methodology enables software engineers to utilize Kanren as an installed Prolog-like language. This is particularly valid for clients of Clojure's 'core.logic', which incorporates numerous Prolog-like augmentations. This “down to earth” utilization of Kanren appears to represent most of Karen's utilization in the industry. Software engineers who need to include a learning-based thinking framework to a current application written in Clojure or Python or JavaScript are commonly not keen on revamping their whole application in Prolog. Inserting a little rationale programming language in Clojure or Python is significantly more engaging. An installed Prolog usage would work similarly also for this reason, apparently. I think Kanren has turned out to be well-known as an inserted rationale language as a result of the small, unadulterated, and center usage, alongside the discussions, blog entries, instructional exercises, and other instructive materials that have turned out since 'The Reasoned Schemer' was distributed.

Notwithstanding the utilization of Kanren as a down to business installed rationale programming language comparable in soul to Prolog, Kanren is being utilized for research in “social” programming. That is, recorded as a hard copy programs that act as scientific relations as opposed to numerical capacities. For instance, in a scheme the affix capacity can attach two records, restoring another rundown: the capacity call (add '(a b c) '(d e)) restores the rundown (a b c d e).

We can, in any case, likewise treat annex as a three-place connection instead of as a two-contention work. The call (appendo '(a b c) '(d e) Z) would then connect the rationale variable Z with the rundown (a b c d e). Obviously, things get all the more fascinating when we place rationale factors in different positions. The call (appendo X '(d e) '(a b c d e)) partners X with (a b c), while the call (appendo X Y (a b c d e)) partners X and Y with sets of records, that, when attached, are equivalent to (a b c d e). For instance X = (a b) and Y = (c d e) are one such pair of qualities. We can likewise compose (appendo X Y Z), which will create boundlessly numerous triples of records X, Y, and Z with the end goal that affixing X to Y produces Z.

This social variant of annex can be effectively communicated in Prolog, and for sure is appeared numerous Prolog instructional exercises. Practically speaking, progressively complex Prolog projects will in general use, at any rate, a couple of additional legitimate highlights, for example, cut, which hinder the capacity to treat the subsequent program as a connection. Interestingly, Kanren is expressly intended to help this style of social programming. Later forms of Kanren have support for representative imperative comprehending (symbolo, numbero, absento, disequality limitations, ostensible rationale programming) to make it simpler to compose non-trifling projects as relations. By and by I never utilize any of the additional intelligent highlights of Kanren, and I compose the majority of my Kanren programs as relations. The most intriguing social projects are the social mediators for a subset of Scheme. These mediators have many intriguing capacities, for example, producing a million Scheme programs that assess to the rundown (I adore you), or inconsequentially creating quines (programs that assess to themselves).

Kanren makes various exchanges offs to empower this social style of programming, which are altogether different from the exchange offs Prolog makes. After some time Kanren has included progressively emblematic requirements, truly turning into an emblematically situated Constraint Logic Programming language. Much of the time these representative limitations make it viable to abstain from utilizing additional consistent administrators like condu and venture. In different cases, these representative imperatives are not adequate. Better help for emblematic requirements is one dynamic zone of Kanren inquire about, alongside the more extensive inquiry of how to compose bigger and progressively complex projects as relations.

Example 1: Python Logic Programming Example:

With logic programming, we are able to compare expressions and resolve unknown values. Consider the following piece of code:

>>> from kanren import run,var,fact

>>> from kanren.assoccomm import eq\_assoccomm as eq

>>> from kanren.assoccomm import commutative,associative

>>> add='add' #Defining operations

>>> mul='mul'

>>> fact(commutative,mul) #Addition and multiplication are commutative and associative

>>> fact(commutative,add)

>>> fact(associative,mul)

>>> fact(associative,add)

>>> a,b,c=var('a'),var('b'),var('c') #Defining variables

>>> #2ab+b+3c is the expression we have'

>>> expression=(add, (mul, 2, a, b), b, (mul, 3, c))

>>> expression=(add,(mul,3,-2),(mul,(add,1,(mul,2,3)),-1)) #Expression

>>> expr1=(add,(mul,(add,1,(mul,2,a)),b),(mul,3,c)) #Expressions to match

>>> expr2=(add,(mul,c,3),(mul,b,(add,(mul,2,a),1)))

>>> expr3=(add,(add,(mul,(mul,2,a),b),b),(mul,3,c))

>>> run(0,(a,b,c),eq(expr1,expression)) #Calls to run()

((3, -1, -2),)

>>> run(0,(a,b,c),eq(expr2,expression))

((3, -1, -2),)

>>> run(0,(a,b,c),eq(expr3,expression))

( )

You’ll see that the third expression gives us nothing. It is mathematically the same, but structurally different.

Example 2: Checking for Prime Numbers in Python Logic Programming:

If we have a list of numbers, we can find out which ones are prime and also generate such numbers. Let’s see how.

>>> from kanren import isvar,run,membero

>>> from kanren.core import success,fail,goaleval,condeseq,eq,var

>>> from sympy.ntheory.generate import prime,isprime

>>> import itertools as it

>>> def prime\_test(n): #Function to test for prime

if isvar(n):

return condeseq([(eq,n,p)] for p in map(prime, it.count(1)))

else:

return success if isprime(n) else fail

>>> n=var() #Variable to use

>>>set(run(0,n,(membero,n,(12,14,15,19,21,20,22,29,23,30,41,44,62,52,65,85)),(prime\_test,n)))

>>> run(7,n,prime\_test(n))

(2, 3, 5, 7, 11, 13, 17)

So, this was all in Python Logic Programming.