First Order Logic

In the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic.

- **"Some humans are intelligent", or**
- **"Sachin likes cricket."**

To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

First-Order logic:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:

    - **Objects:** It refers to an entity that exists in the real world. **For example,** Ram, John, etc. are referred to as Objects.
    - **Relations:** The relation of an object with the other object defines its relation**. For example,** brother, mother, king, etc. are some types of relations which exist in the real world.

- **Functions:** Any function performed by the object/on the object. **For example** writes, eats, etc. are some of the functions.

. Syntax of First-Order logic:

The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2, ……, term n)**.
- A term is a logical expression that refers to an object.

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).
Chinky is a cat: => cat (Chinky).

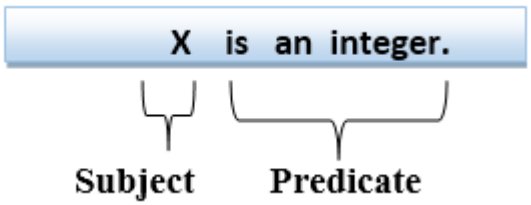
Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:

1. **Universal Quantifier, (for all, everyone, everything)**

1. Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀.

Note: In universal quantifier we use implication "→".

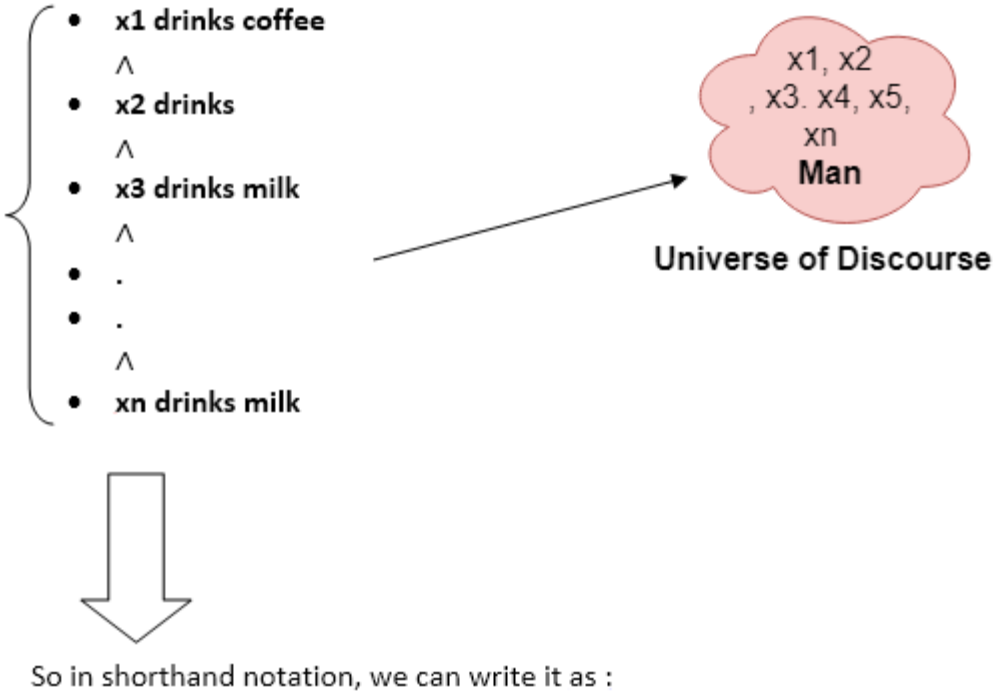
f x is a variable, then ∀x is read as:

- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

Let a variable x



x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

2. Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃,

> ***Note: In Existential quantifier we always use AND or Conjunction symbol (∧).***

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some students are intelligent.

x: student(x) intelligent(x)

It will be read as: There are some x where x is a student who is intelligent.

3. ## Nested Quantifiers:

   It is the nesting of the same type of quantifier. One predicate is nested under the other predicate. Two quantifiers are nested if one is within the scope of the other.

   Ex1:Everybody loves somebody

   Example ∃y ∀x Loves(x, y)

   For every person, there is someone that person loves

   Ex2: Brothers are siblings

   ∀x ∀y brother(x,y) →Sibling(x,y)

4. **Equality:** We use the equality symbol to express that two terms refer to the same object**. For example,** Eleveenth_President(India)= Dr. APJ Abdul Kalam. Here, both LHS is equal to RHS. It means that both terms refer to the same entity/ person.

   Equality can also be used with the negation to insist that two terns are not the same object.

Richard has at least two brothers

$\exists x,y$ Brother(x, Richard) ∧ Brother (x, Richard) ∧ ¬(x==y)

Some Examples of FOL using quantifier:

1. All birds fly.
In this question the predicate is "**fly(bird)**."
And since there are all birds who fly so it will be represented as follows.

x bird(x) →fly(x).

2. Every man respects his parent.
In this question, the predicate is "**respect(x, y)," where x=man, and y= parent**.
Since there is every man so will use ∀, and it will be represented as follows:

x man(x) → respects (x, parent).

3. Some boys play cricket.
In this question, the predicate is "**play(x, y)**," where x= boys, and y= game. Since there are some boys so we will use **, and it will be represented as**:

x boys(x) → play (x, cricket).

4. Not all students like both Mathematics and Science.
In this question, the predicate is "**like(x, y)," where x= student, and y= subject**.
Since there are not all students, so we will use **with negation, so** following representation for this:

¬ (x) [student(x) → like(x, Mathematics) like(x, Science)].

Some more examples:

Example 1: Lipton is a tea.
Solution: Here, the object is Lipton.
It will be represented as **Tea(Lipton).**
Note: In this example, there is no requirement of quantifiers because the quantity is not specified in the given predicate.

Example 2: Every man is mortal.
Solution: Here, the quantifier is the universal identifier, and the object is man.

Let x be the man.
Thus, it will be represented as **x: man(x) → mortal(x).**

Example 3: All girls are beautiful.
Solution: Here, we are talking about all girls. It means universal quantifier will be used. The object is girls. Let, y be the girls.
Therefore, it will be represented as **y:girls(y) → beautiful(y).**

Example 4: All that glitters is not gold.
Solution: Here, we will represent gold as x.
Therefore, it will be represented as **∀x: glitters(x) → ¬gold(x).**

Example 5: Some boys are obedient.
Solution: Here, boys are objects. The quantifier used will be existential quantifier. Let x be the boys. Thus, it will be represented as
∃x: boys(x) → obedient(x).

Example 6: Some cows are black and some cows are white.
Solution: Let, x be the cows. Therefore, it will be represented as:
∃x: cows(x) → black(x) ∧ white(x).

Example 6: All kings are person:

Solution: $\forall x\ King(x) \Rightarrow Person(x)$

For all x, if x is a king, then x is a person

Some more examples:

1. "Every house is a physical object" is translated as

∀x house(x) → physical object(x), where house and physical object are unary predicate symbols.

2. "Some physical objects are houses"'

is translated as ∃x.(physical object(x) ∧ house(x))

3. "every house is owned by somebody" is translated as
   ∀x ∃y. (house(x) → owns(y, x)),

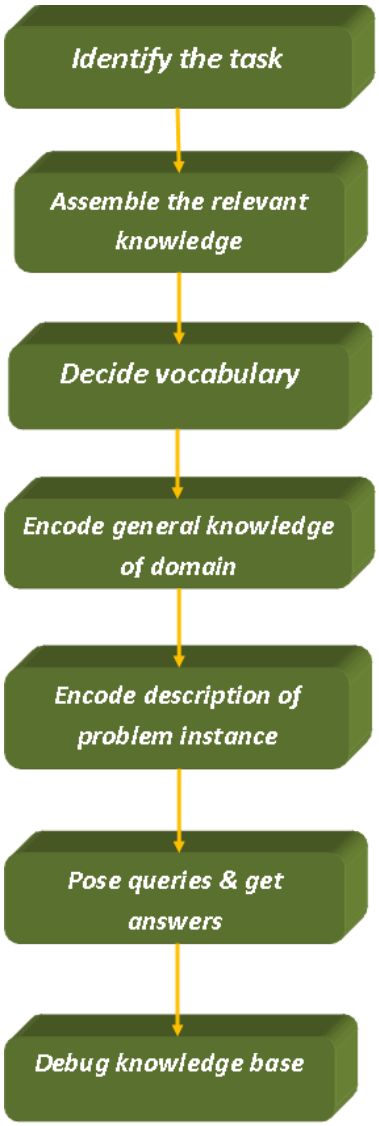
4. Some Dogs bark
   ∃x.(dog(x) ∧ bark(x))

Knowledge Engineering in FOPL

Knowledge engineering is the process where a knowledge engineer investigates a specific domain, learn the important concepts regarding that domain, and creates the formal representation of the objects and relations in that domain.

Knowledge Engineering Process

An engineering term is used when we are talking about any project. **Therefore, knowledge engineering over a project involves the below described steps:**



- **Identify the task:** A knowledge engineer should be able to identify the task by asking a few questions like:
- Do the knowledge base will support?
- What kinds of facts will be available for each specific problem?

The task will identify the knowledge requirement needed to connect the problem instance with the answers.

Example: For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location?

- **Assemble the relevant knowledge:** A knowledge engineer should be an expert in the domain. If not, he should work with the real experts to extract their knowledge. This concept is known as **Knowledge Acquisition**.

Note: Here, we do not represent the knowledge formally. But to understand the scope of the knowledge base and also to understand the working of the domain.

Example: For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify.

- **Decide on a vocabulary of constants, predicates, and functions:** Translating important domain-level concepts into logical level concepts.

Here, the knowledge engineer asks questions like:

- What are the elements which should be represented as objects?
- What functions should be chosen?

After satisfying all the choices, the vocabulary is decided. It is known as the **Ontology of the domain**. Ontology determines the type of things that exists but does not determine their specific properties and interrelationships.

For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time?

- **Encode general knowledge about the domain:** In this step, the knowledge engineer pen down the axioms for all the chosen vocabulary terms.

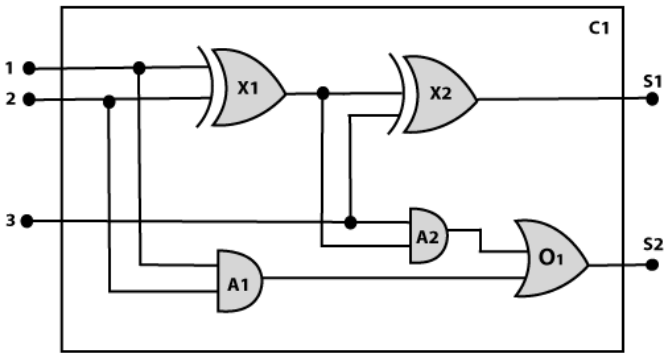
- **Encode description of the specific problem instance:** We write the simple atomic sentences for the selected vocabulary terms. We encode the chosen problem instances.

  For a logical agent, problem instances are supplied by the sensors.

- **Raise queries to the inference procedure and get answers:** It is the testing step. We apply the inference procedure on those axioms and problem-specific facts which we want to know.
- **Debug the knowledge base:** It is the last step of the knowledge engineering process where the knowledge engineer debugs all the errors.

The electronic circuits domain:

Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (**One-bit full adder**) which is given below



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks.

At the first level or highest level, we will examine the functionality of the circuit:

- **Does the circuit add properly?**
- **What will be the output of gate A2, if all the inputs are high?**

At the second level, we will examine the circuit structure details such as:

- **Which gate is connected to the first input terminal?**
- **Does the circuit have feedback loops?**

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: **AND, OR, XOR, and NOT**.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate(X1)**. The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit (C1)**.

For the terminal, we will use predicate: **Terminal(x)**.

For gate input, we will use the function **In(1, X1)** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, X1)**.

The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.

The connectivity between gates can be represented by predicate **Connect(Out(1, X1), In(1, X2))**.

We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

- If two terminals are connected then they have the same input signal, it can be represented as:

∀ t1, t2 Terminal (t1) ∧ Terminal (t2) ∧ Connect (t1, t2) → Signal (t1) = Signal (t2).

- Signal at every terminal will have either value 0 or 1, it will be represented as:

∀ t Terminal (t) →Signal (t) = 1 ∨Signal (t) = 0.

- Connect predicates are commutative:
  ∀ t1, t2 Connect(t1, t2) → Connect (t2, t1).

- Output of AND gate will be zero if and only if any of its input is zero.

∀ g Gate(g) ∧ Type(g) = AND →Signal (Out(1, g))= 0 ⇔ ∃n Signal (In(n, g))= 0.

- Output of OR gate is 1 if and only if any of its input is 1:

1.

∀ g Gate(g) ∧ Type(g) = OR → Signal (Out(1, g))= 1 ⇔ ∃n Signal (In(n, g))= 1

- Output of XOR gate is 1 if and only if its inputs are different:

1.

∀ g Gate(g) ∧ Type(g) = XOR → Signal (Out(1, g)) = 1 ⇔ Signal (In(1, g)) ≠ Signal (In(2, g)).

- Output of NOT gate is invert of its input:

1. ∀ g Gate(g) ∧ Type(g) = NOT → Signal (In(1, g)) ≠ Signal (Out(1, g)).

- All the gates in the above circuit have two inputs and one output (except NOT gate).

1. ∀ g Gate(g) ∧ Type(g) = NOT → Arity(g, 1, 1)
2. ∀ g Gate(g) ∧ r =Type(g) ∧ (r= AND ∨r= OR ∨r= XOR) → Arity (g, 2, 1).

- All gates are logic circuits:

1. ∀ g Gate(g) → Circuit (g).

5. Encode a description of the problem instance:

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below:

Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

1. For XOR gate: Type(x1)= XOR, Type(X2) = XOR
2. For AND gate: Type(A1) = AND, Type(A2)= AND
3. For OR gate: Type (O1) = OR.

And then represent the connections between all the gates.

Note: Ontology defines a particular theory of the nature of existence.

6. Pose queries to the inference procedure and get answers:

In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

1. ∃ i1, i2, i3 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2 ∧ Signal (In(3, C1))= i3
2. ∧ Signal (Out(1, C1)) =0 ∧ Signal (Out(2, C1))=1

7. Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.

In the knowledge base, we may have omitted assertions like 1 ≠ 0.

Inference in First-Order Logic

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

FOL inference rules for quantifier:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- **Universal Generalization**
- **Universal Instantiation**
- **Existential Instantiation**
- **Existential introduction**

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "**A byte contains 8 bits**", so for **x P(x)** "**All bytes contain 8 bits**.", it will also be true.

2. Universal Instantiation:

- As per UI, **we can infer any sentence obtained by substituting a ground term for the variable**.

- The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from **x P(x) for any object in the universe of discourse**.
- It can be represented as: $\frac{\forall x\, P(x)}{P(c)}$.

Example:1.

IF "Every person like ice-cream"=> ∀x P(x) so we can infer that
"John likes ice-cream" => P(c)

Example: 2.

Every man is mortal.

It is represented as ∀ x: man(x) → mortal(x).
In UI, we can infer different sentences as:
man(John) → mortal(John)
man(Aakash) → mortal(Aakash), etc.

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- Notation: Let, the variable be v which is replaced by a constant symbol k for any sentence α. The value of k is unique as it does not appear for any other sentence in the knowledge base. Such type of constant symbols are known as Skolem constant. As a result, EI is a special case of Skolemization process.

It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

For example: ∃x:steal(x, Money).
We can infer from this: **steal(Thief, Money)**

- The above used K is a constant symbol, which is called **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$
- **Example: Let's say that,**
  "Priyanka got good marks in English."
  "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

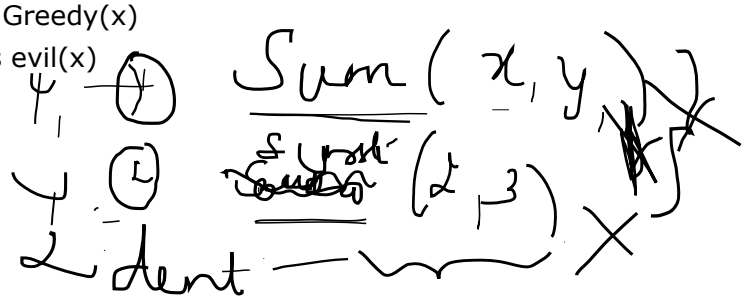
$$\frac{p1', p2', \ldots, pn', (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

t is a lifted version of Modus Ponen as it uplifts the Modus Ponens from ground propositions to FOPL. Generalized Modus Ponen is more generalized than Modus Ponen. It is because, in generailzed, the known facts and the premise of the implication are matched only upto a substitution, instead of its exact match.

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

1. Here let say, p1' is king(John) p1 is king(x)
2. p2' is Greedy(y) p2 is Greedy(x)
3. θ is {x/John, y/John} q is evil(x)
4. SUBST(θ,q).



Unification

What is Unification?

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and be a unifier such that, $\Psi_1 = \Psi_2$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.

Example1: Find the MGU for Unify{King(x), King(John)}

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g.2. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)..... (i)
P(a, f(z))..... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if $\Psi_1$is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return { (Ψ_2/ Ψ_1)}.

c) Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE,

b. Else return {(Ψ_1/ Ψ_2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST

Examples

1. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}
S_0 => {prime(11) , prime(y)}
SUBST θ= {11/y}

S_1 => {prime(11) , prime(11)} , **Successfully unified.**

Unifier: {11/y}.

2.UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)
S_0 => { knows(Richard, x); knows(Richard, John)}

SUBST θ= {John/x}
S_1 => { knows(Richard, John); knows(Richard, John)}, **Successfully Unified. Unifier: {John/x}.**