Program7:
Implement unification in first order logic

Algorithm:

First Order logic:

1. John is a human
   → human (John)

2. Every human is a mortal
   ∀x (human(n) ⟶ mortal(n))

3. John loves Mary
   → loves (John, Mary).

4. There is someone who loves Mary.
   → ∃ n(loves (n, May))

5. All dogs are animals
   ∀x (dogs(n) ⟶ animal (n))

6. Some dogs are brown
   ∃x (dog (n) ^ brown (n)).

7. There is no person who is both a bachelor and married.
   ∀x(married (n) ⟶ ¬ bachelor(n))

Pseudocode:

```
Function Translate-to-sol(sentence):
    sentence = sentence.strip().lower()
    if "is both" in sentence and "and" in sentence:
        return bachelor_and_married (sentence)

    if "is the mother of" in sentence:
        return mother of (sentence)

    if "are both students" in sentence:
        return both_students (sentence)

    if "if" in sentence and "then" in sentence:
        return if_then (sentence)

    if "there is a person" in sentence:
        return knows - everyone (sentence)

    if "taller than themselves" in sentence
        return nobody_taller (sentence)

    return "Translation not available"

def mother-of (sentence):
    match = re.match (r"[a-z A-Z]+) is
        the mother of ([a-z A-Z]+)", sentence)
    if match:
        subject = match.group(1)
        obj = match.group(2)
        return if "Mother of ({subject}, {obj})
    return "Invalid sentence".
```

Unification of expression:

```
def unify (expr1, expr2, subs):
    if subs is null:
        subs = { }.
    if expr1 == expr2
        return subs
    if is-var (expr1):
        return unify_var(expr1, expr2, subs)
    if is_var(expr2):
        return unify_var( expr2, expr1, subs)
    if is_comp (expr1) and is_comp(expr2):
        if expr1[0] != expr2[0] or len (expr1)
                != len (expr2):
            raise error
        return unify_lists (expr1, expr2,
                unify. (expr1[0], expr2[0], subs))
    raise error

def unify_var( var, expr, subs):
    if var in subs
        return unify ( subs[var], expr, subs)
    elif occur_check (var, expr, subs):
        raise error
    else:
        subs[var] = expr
        return subs

def unify_list (L1, L2, subs)
    for exp1, expr2 in zip (L1, L2),
        sub = unify (expr1, expr2, subs)
    return subs
```

Output:

1. Mary is the mother of John
   → Mother Of (Mary, John)

2. John and Mary are both students.
   → student (John) ^ student (Mary)

3. If it raining then ground is wet.
   → Raining → wet (ground)

4. Nobody is taller than themselves
   → ∀x ^Taller (x, x).

```
def is_var(term)
    return isinstance(term, str) and term[0].islower()

def is_comp(term):
    return is_instance (term, list, tuple)) and
        len(term) >0 .

def occur-check (var, expr, subs):
    if var == expr:
        return true
    elif is_comp (expr):
        return any (occur-check (var, sub,
            subs) for sub in expr).

    elif expr in subs
        return occurs_check (var, subs[expr]
            subs)
    return false.
```

Output:

Expression f(x g(y)) & f(a, g(b))
Input : ("f", 'x', ('g', 'y'))
('f', 'a', ('g', 'b'))
substitutions : {'x' : 'a', 'y' : 'b'}

Prove using forward chaining : technique :

Facts :

1. Ravi enjoy a wide variety of foods :

FOL $\forall x$ Food (x) $\longrightarrow$ Enjoys (Ravi, x)

CNF $\neg$ Food (x) $\vee$ Enjoys (Ravi, x)

2. Bananas are food.

FOL Food (Banana)
CNF : Food (Banana)

3. Pizza is food

FOL = CNF : Food (Pizza)

4. A food is anything that anyone consumes and isn't harmed by.

FOL: $\forall x (\exists y$ (consumes (y, x) $\wedge \neg$ harmedby (y, x))
$\Rightarrow$ Food (x)

CNF: $\neg \exists y$ (consumed (y, x) $\wedge \neg$ Harmed by (y, x)) (or)
$\vee$ Food (x)

Subst y with c(x)

$\neg$ consumed (c(x), x) $\vee$ HarmedBy (C(x), x) $\vee$ food(x)

5. Sam eats Idli and is still alive

FOL / CNF : Consumes (Sam, idli) $\wedge \neg$ Harmed By(Sam, idli)

6. Gill eats everything sam eats

FOL : $\forall x$ (consumes (sam, x) $\rightarrow$ consumes (Bill, x))

CNF : $\neg$ consumes (sam, x) $\vee$ consumes (Bill, x)

Goal : Ravi like Idli ; Enjoys (Ravi, idli)

The Proof tree :

Enjoy (Ravi, Idli)

Food (Idli)

Food (Pizza) consumes (sam, Idli) $\neg$HarmedBy(sam, Idli)

Code:

```
def unify_terms(term_a, term_b, subs=None):
    if subs is None:
        subs = {}

    if term_a == term_b:
        return subs

    if is_variable(term_a):
        return unify_with_var(term_a, term_b, subs)
    if is_variable(term_b):
        return unify_with_var(term_b, term_a, subs)

    if is_compound(term_a) and is_compound(term_b):
        if term_a[0] != term_b[0] or len(term_a[1]) != len(term_b[1]):
            return None
        for subterm_a, subterm_b in zip(term_a[1], term_b[1]):
            subs = unify_terms(subterm_a, subterm_b, subs)
            if subs is None:
                return None
        return subs

    if isinstance(term_a, list) and isinstance(term_b, list):
        if len(term_a) != len(term_b):
            return None
        for element_a, element_b in zip(term_a, term_b):
            subs = unify_terms(element_a, element_b, subs)
            if subs is None:
                return None
        return subs

    return None
def unify_with_var(var, expr, subs):

    if var in subs:
        return unify_terms(subs[var], expr, subs)
    if expr in subs:
        return unify_terms(var, subs[expr], subs)
    if occurs_check(var, expr, subs):
        return None  # Cyclic substitution check failed
    subs[var] = expr
    return subs


def occurs_check(var, expr, subs):

    if var == expr:
```

```python
        return True
    if is_compound(expr):
        return any(occurs_check(var, arg, subs) for arg in expr[1])
    if isinstance(expr, list):
        return any(occurs_check(var, item, subs) for item in expr)
    if expr in subs:
        return occurs_check(var, subs[expr], subs)
    return False


def is_variable(item):

    return isinstance(item, str) and item.startswith('?')


def is_compound(item):

    return isinstance(item, tuple) and len(item) == 2 and isinstance(item[1], list)


if __name__ == "__main__":
    print("Enter expressions in the following format:")
    print("Compound terms: ('f', ['a', 'b'])")
    print("Variables: '?x', '?y'")
    print("Lists: ['a', 'b']")
    print("Constants: 'a', 'b', etc.\n")

    term_1 = eval(input("Enter the first expression (Ψ₁): "))
    term_2 = eval(input("Enter the second expression (Ψ₂): "))

    substitution_result = unify_terms(term_1, term_2)
    if substitution_result is None:
        print("Unification failed!")
    else:
        print("Unification successful!")
        print("Substitution Set:", substitution_result)
```

Output Snapshot:

```
Enter expressions in the following format:
Compound terms: ('f', ['a', 'b'])
Variables: '?x', '?y'
Lists: ['a', 'b']
Constants: 'a', 'b', etc.

Enter the first expression (Ψ₁): ('Studies',['Abubakar','?x'])
Enter the second expression (Ψ₂): ('Studies',['?y','AI'])
Unification successful!
Substitution Set: {'?y': 'Abubakar', '?x': 'AI'}
```