WEEK 07 - FIRST ORDER LOGIC OF UNIFICATION

```
import ast
from typing import Union, List, Dict, Tuple
from collections import deque

# Define Term Classes
class Term:
    def substitute(self, subs: Dict[str, 'Term']) -> 'Term':
        raise NotImplementedError

    def occurs(self, var: 'Variable') -> bool:
        raise NotImplementedError

    def __eq__(self, other):
        raise NotImplementedError

    def __str__(self):
        raise NotImplementedError

class Variable(Term):
    def __init__(self, name: str):
        self.name = name

    def substitute(self, subs: Dict[str, Term]) -> Term:
        if self.name in subs:
            return subs[self.name].substitute(subs)
        return self

    def occurs(self, var: 'Variable') -> bool:
        return self.name == var.name

    def __eq__(self, other):
        return isinstance(other, Variable) and self.name == other.name

    def __str__(self):
        return self.name

class Constant(Term):
    def __init__(self, name: str):
```

```
        self.name = name

    def substitute(self, subs: Dict[str, Term]) -> Term:
        return self

    def occurs(self, var: 'Variable') -> bool:
        return False

    def __eq__(self, other):
        return isinstance(other, Constant) and self.name == other.name

    def __str__(self):
        return self.name

class Function(Term):
    def __init__(self, name: str, args: List[Term]):
        self.name = name
        self.args = args

    def substitute(self, subs: Dict[str, Term]) -> Term:
        substituted_args = [arg.substitute(subs) for arg in self.args]
        return Function(self.name, substituted_args)

    def occurs(self, var: 'Variable') -> bool:
        return any(arg.occurs(var) for arg in self.args)

    def __eq__(self, other):
        return (
            isinstance(other, Function) and
            self.name == other.name and
            len(self.args) == len(other.args) and
            all(a == b for a, b in zip(self.args, other.args))
        )

    def __str__(self):
        return f"{self.name}({', '.join(str(arg) for arg in self.args)})"

def parse_expression(expr: List) -> Term:
    if not isinstance(expr, list) or not expr:
        raise ValueError("Expression must be a non-empty list")
```

```
    func_name = expr[0]
    args = expr[1:]

    parsed_args = []
    for arg in args:
        if isinstance(arg, list):
            parsed_args.append(parse_expression(arg))
        elif isinstance(arg, str):
            if arg[0].islower():
                parsed_args.append(Variable(arg))
            elif arg[0].isupper():
                parsed_args.append(Constant(arg))
            else:
                raise ValueError(f"Invalid argument format: {arg}")
        else:
            raise ValueError(f"Unsupported argument type: {arg}")

    return Function(func_name, parsed_args)

def unify_terms(term1: Term, term2: Term) -> Union[Dict[str, Term], str]:
    # Initialize substitution set S
    S: Dict[str, Term] = {}

    # Initialize the equation list
    equations: deque[Tuple[Term, Term]] = deque()
    equations.append((term1, term2))

    while equations:
        s, t = equations.popleft()
        s = s.substitute(S)
        t = t.substitute(S)

        if s == t:
            continue
        elif isinstance(s, Variable):
            if t.occurs(s):
                return "FAILURE"
            S[s.name] = t
            # Apply the substitution to existing substitutions
```

```
            for var in S:
                S[var] = S[var].substitute({s.name: t})
        elif isinstance(t, Variable):
            if s.occurs(t):
                return "FAILURE"
            S[t.name] = s
            for var in S:
                S[var] = S[var].substitute({t.name: s})
        elif isinstance(s, Function) and isinstance(t, Function):
            if s.name != t.name or len(s.args) != len(t.args):
                return "FAILURE"
            for s_arg, t_arg in zip(s.args, t.args):
                equations.append((s_arg, t_arg))
        elif isinstance(s, Constant) and isinstance(t, Constant):
            if s.name != t.name:
                return "FAILURE"
            # else, they are equal; continue
        else:
            return "FAILURE"

    return S

def format_substitution(S: Dict[str, Term]) -> str:
    if not S:
        return "{}"
    return "{ " + ", ".join(f"{var} = {term}" for var, term in S.items())
+ " }"

def unify(expression1: List, expression2: List) -> Union[Dict[str, Term],
str]:
    try:
        term1 = parse_expression(expression1)
        term2 = parse_expression(expression2)
    except ValueError as e:
        return f"FAILURE: {e}"

    result = unify_terms(term1, term2)

    return result
```

```
def main():
    print("=== Unification Algorithm ===\n")
    print("Please enter the expressions in list format.")
    print("Example: [\"Eats\", \"x\", \"Apple\"]\n")

    try:
        expr1_input = input("Enter Expression 1: ")
        expression1 = ast.literal_eval(expr1_input)
        if not isinstance(expression1, list):
            raise ValueError("Expression must be a list.")

        expr2_input = input("Enter Expression 2: ")
        expression2 = ast.literal_eval(expr2_input)
        if not isinstance(expression2, list):
            raise ValueError("Expression must be a list.")

    except (SyntaxError, ValueError) as e:
        print(f"Invalid input format: {e}")
        return

    result = unify(expression1, expression2)

    if isinstance(result, str):
        print("\nUnification Result:")
        print(result)
    else:
        print("\nUnification Successful:")
        print(format_substitution(result))

if __name__ == "__main__":
    main()
print("Vaibhav Urs A N")
print("1BM22CS315\n")
```

OUTPUT

```
=== Unification Algorithm ===

Please enter the expressions in list format.
Example: ["Eats", "x", "Apple"]

Enter Expression 1: ["Eats", "x", "Apple"]
Enter Expression 2: ["Eats", "Banana", "y"]

Unification Successful:
{ x = Banana, y = Apple }
Vaibhav Urs A N
1BM22CS315
```