# Proyecto Parcial 1: Matemáticas Discretas y sus Aplicaciones

## 1. Introducción

El objetivo de este proyecto es implementar un evaluador de fórmulas en Lógica Proposicional (LP) utilizando el lenguaje de programación Python. Dada una fórmula en LP, el sistema debe determinar si es una tautología, una contradicción o una contingencia. La entrega se divide en dos partes:

- **a) Función de valoración:** Dada una fórmula y una asignación de valores de verdad a sus atomos, debe devolver el valor de verdad de la fórmula con dicha valoracion. Si la fórmula no es valida, el sistema debe indicarlo.
- **b) Probador de fórmulas:** Dada una fórmula, utilizando la funcion anterior, se deben probar todas las posibles valoraciones y determinar si la fórmula es una tautologıa, contradiccion o contingencia.

## 2. Explicación del código

### Parte A

```
# Parte A

v = dict()

def str_a_list(string: str, ignore: set) -> list:
    cars = []
    for atomo in string:
        if atomo not in ignore:
            cars.append(atomo)
    return cars
```

Primeramente, se inicializa el diccionario `v = dict()` en el cual se guardarán los valores de verdad de los átomos de las fórmulas que vayamos a ingresar. Este diccionario servirá tanto para la parte A como para la parte B, en donde sus valores irán actualizandose para probar todas las posibles valoraciones de las fórmulas ingresadas y determinar si son tautologías, contradicciones o contingencias. Más adelante se entrará en detalle sobre esa parte.

Por otro lado, se define la función `str_a_list()` la cual recibe como parámetros a la fórmula ingresada como `string` y al conjunto de símbolos o caracteres a omitir ( `ignore: set` ) en la iteración que se realiza para extraer cada caracter de la cadena/fórmula. La flecha `-> list` significa que se espera que la función devuelva un valor `list` .

Dentro de la función, se inicializa la lista `cars` que va a ser el conjunto de cada caracter que tiene la fórmula ingresada (e.g. `['(', 'p', '&', 'q', ')']` ), ignorando los símbolos específicados en el conjunto `ignore` que en nuestro caso van a ser los espacios en blanco ( `{' '}` ).

En el bucle `for atomo in string:` , `atomo` recorre cada caracter de la cadena `string` y si no está dentro del conjunto especificado `ignore` , agrega `atomo` a la lista de caracteres `cars` .

```
def valoracion(formula, der: int, izq: int):
    resultado = -1
    if der == izq:
        if formula[der] in v:
            resultado = v[formula[der]]
```

```
        elif formula[der] == '!':
            resultado = 1 - valoracion(formula, der + 1, izq)
        elif formula[der] == '(' and formula[izq] == ')':
            count, i, oper = 0, der + 1, -1
            while i < izq and oper == -1:
                if formula[i] == '(':
                    count += 1
                elif formula[i] == ')':
                    count -= 1
                if formula[i] in {'&', '|'} and count == 0:
                    oper = i
                i += 1

            if oper != -1:
                izquierda = valoracion(formula, der + 1, oper - 1)
                derecha = valoracion(formula, oper + 1, izq - 1)
                if izquierda != -1 and derecha != -1:
                    if formula[oper] == '&':
                        resultado = izquierda and derecha
                    elif formula[oper] == '|':
                        resultado = izquierda or derecha
            else:
                resultado = -1
        return resultado
```

La función `valoración()`, encargada de la lógica de la valoración de las fórmulas, recibe como parámetros a `formula`, que va a ser la lista de caracteres de una fórmula dada (en el main() se utiliza la función str_a_list() para esto), y los índices de tipo int der e izq. Estos dos son los que indican el índice que se va a recorrer de izquierda a derecha y de derecha a izquierda, respectivamente.

Inicialmente se inicializa la variable resultado con un valor de -1, ya que esta va a indicar el valor de verdad (0 o 1) de las fórmulas dependiendo de la validez y si se redactaron bien o no, por lo tanto la función termina devolviendo ese valor inicial si no se cumplen con estos requisitos. Luego se indica que si el valor del índice inferior (der) es igual al del índice superior (izq), si el caracter en el índice der de la lista formula, se encuentra en el diccionario v, la variable `resultado` toma el valor de `formula[der]`. Sin embargo, si la función aún no ha llegado a ese punto y el caracter `formula[der]` es `!`, se aplica recursivamente la función valoración() a lo que sigue después del símbolo, es decir, desde el index `der + 1` hasta `izq`, para posteriormente "negar" su valor restandole a 1 la valoración de ese fragmento (e.g. si su valor es 1 su negación sería `1 - 1 = 0`, y viceversa `1 - 0 = 1`)

Ahora, para lograr esto y reducir cada fórmula a sus expresiones más básicas (átomos), el siguiente bloque `elif formula[der] == '(' and formula[izq] == ')':` se va a encargar de hayar el operador principal u operador medio (`oper`) que va a permitir separar el código en expresiones cada vez más simples desde donde usando los valores de verdad de cada átomo se puedan ir construyendo los valores de verdad de cada expresión separada por un AND (`&`) o un OR (`|`).

La manera en que el programa identifica esto es por medio de las variables `count, i, oper`, que inician con un valor de `0`, `der + 1` y `-1`, respectivamente. La variable `count` es la que permite hayar el operador medio, ya que va a sumar los paréntesis iniciales y finales como 1 y -1, lo cual hace que el iterador `i` que recorre toda la formula de izquierda a derecha desde el índice indicado, pueda hacer esta cuenta. Si `count == 0` significa que el operador que sigue es el principal, ya que separa la fórmula en dos expresiones distintas más simples. Por lo tanto, si el iterador `i` se para sobre un símbolo `&` o `|` con `count` en 0, la variable `oper` que representa el operador medio cuyo valor inicial es `-1` por si la fórmula está mal redactada, toma el valor de `i`.

Si esto se cumple, `oper` va a ser distinto de `-1`, por lo que el bloque `if oper != -1:` se ejecuta. Aquí pasa lo que queríamos lograr: que las dos partes separadas por el operador medio se puedan evaluar independientemente y aplicarseles el mismo ciclo desde sus partes atómicas y construir los valores de verdad de cada parte hasta poder operar ambas expresiones separadas por `oper`. Entonces si el operador medio es `&` se opera como `izquierda and derecha` (los dos lados que divide `oper`), y de igual manera con `|`: `izquierda or derecha`; esto para que el valor de dicha operación sea guardado en la variable resultado que es el valor que devuelve la función (`0` o `1`)

En cambio, si `oper == -1` significa que algo en la formulación de la expresión no es válido, por lo tanto devuelve el valor de resultado como `-1`; lo cual significa que la fórmula no es válida.

```
def main_a():
    print("Parte A\n")
    N = int(input())
    if 1 <= N <= 5:
        for j in range(N):
            entrada = input().split()
            if len(entrada) == 2 and entrada[1] in {'0', '1'}:
                v[entrada[0]] = int(entrada[1])
            else:
                print("Valor inválido")
    else:
        print("Fuera de rango (1 - 5)")

    S = int(input())
    if 1 <= S <= 100:
        for k in range(S):
            formula = str_a_list(input(), {' '})
            resultado = valoracion(formula, 0, len(formula) - 1)
            print("-1" if resultado == -1 else "1" if resultado else "0")
    else:
        print("Fuera de rango (1 - 100)")

main_a()
```

Este bloque de código ya es el encargado de manejar toda la lógica del programa, llamando las funciones definidas previamente para determinar el valor de verdad de cada fórmula ingresada con los valores de los átomos también proporcionados por el usuario.

Para empezar, la variable `N` va a guardar un valor entero que el usuario ingrese por teclado en la terminal, y este número es el que representa el número de átomos que pueden tener las fórmulas, dentro de un rango de 1 <= N <= 5. Si `N` se encuentra dentro del rango, el sistema va a permitir ingresar `N` entradas que van a ser los átomos seguidos por su valor de verdad separados por un espacio. Estas entradas se van a guardar como una lista `entrada` por el método `.split()` aplicado al `input()`; por ende si se ingresa e.g. `p 0`, `entrada` guardaría los valores así: `entrada = ['p', '1']`. Ahora, si `entrada` guarda dos valores, o su longitud es 2, y su segundo valor (en el índice `entrada[1]`) es 0 ó 1, va a guardar en el diccionario inicial `v` que almacena los valores de verdad de los átomos, el átomo ingresado como la clave y el valor asignado como su valor (como un entero) de la forma `{'atomo': int('valor')}` (e.g. `{'p': 0}`).

Si `N` no está en el rango, el sistema indica que está fuera de rango, y si la forma en la que ingresa los átomos no es de la forma `p 0` también indica que ingreso un valor no válido.

Una vez ingresados los átomos con sus respectivos valores de verdad, se ingresa el valor `S` que indica el número de fórmulas a evaluar. De la misma forma que `N`, dentro del rango `1 <= S <= 100` el sistema va a permitir ingresar las `S` fórmulas que van a ser guardadas como listas con cada elemento como cada uno de los caracteres de la expresión

omitiendo los espacios en blanco ({' '}); esto gracias a la función previamente definida `str_a_list()`. Por último, con cada fórmula ingresada se aplica la función `valoracion()` que va a tener como su índicie inferior `der` a 0 y su límite superior `izq` como la longitud de `formula`, para que pueda entrar dentro del rango de iteración por su inicio en 0. Ya finalmente, el sistema imprime los valores `1` si `resultado` es `True`. `0` si `resultado` es `False`, y `-1` si `resultado` es `-1`, es decir, que la fórmula no es válida.

## Parte B

Esta parte del proyecto se basa en la implementación de un sistema que evalúa si una fórmula en lógica proposicional es una tautología, una contradicción o una contingencia. Para lograr esto, se utiliza la función de valoración desarrollada en la Parte A, probando todas las posibles asignaciones de valores de verdad a los átomos que aparecen en la fórmula. La razón detrás de esta implementación es garantizar que cada expresión lógica ingresada sea analizada en todos los escenarios posibles para determinar su naturaleza con certeza. De esta manera, el sistema podrá clasificar cualquier fórmula dentro de estas tres categorías basándose en el resultado de su evaluación en múltiples casos.

El primer paso para lograr esto es identificar cuáles son los átomos presentes en la expresión ingresada. Para ello, se utiliza la función `extraer_atomos()`, que recorre la fórmula y almacena en una lista todos los caracteres que no sean operadores lógicos ni paréntesis.

```
def extraer_atomos(expresion):
    atomos = []
    operadores = {'(', ')', '!', '&', '|'}
    for i in expresion:
        if i not in operadores and i not in atomos:
            atomos.append(i)
    return atomos
```

Esto es fundamental, ya que permite definir las variables proposicionales de la fórmula, lo que luego permitirá generar todas las combinaciones de valores de verdad posibles. Sin esta identificación previa, no podríamos realizar la evaluación exhaustiva que requiere el problema, pues no tendríamos claro qué elementos deben recibir diferentes valores de verdad a lo largo del proceso. Es importante recordar que en la Parte A del proyecto ya se hizo un tratamiento similar, en donde se extraían los caracteres de la expresión para estructurar la fórmula como una lista de elementos, lo cual permitía luego evaluarla de manera jerárquica. Aquí aplicamos un principio similar, pero con el objetivo de conocer las variables que deben evaluarse bajo diferentes combinaciones.

Una vez que se han identificado los átomos de la expresión, el siguiente paso es generar todas las posibles combinaciones de valores de verdad que estos pueden tomar. Dado que cada átomo puede tener dos posibles valores (0 o 1), la cantidad total de combinaciones estará determinada por la fórmula `2^n`, donde `n` es el número de átomos distintos en la fórmula.

```
def generar_valores(atomos):
    n_atomos = len(atomos)
    total_combinaciones = 2 ** n_atomos
    combinaciones = []
    for i in range(total_combinaciones):
        combinacion = {}
        num = i
        valores_binarios = [0] * n_atomos
        for j in range(n_atomos - 1, -1, -1):
            valores_binarios[j] = num % 2
            num = num // 2
        for k in range(n_atomos):
            combinacion[atomos[k]] = valores_binarios[k]
```

```
            combinaciones.append(combinacion)
        return combinaciones
```

Para lograr esto, se genera una lista en donde cada elemento es un diccionario que asocia cada átomo con un valor específico. Esto se hace iterando desde `0` hasta `2^n - 1`, convirtiendo cada número en su representación binaria y asignando estos valores a los átomos en cuestión. De esta manera, obtenemos todas las posibles asignaciones que pueden existir para la fórmula dada, permitiéndonos evaluarla en cada uno de estos casos.

```
def evaluador(formula):
    form = str_a_list(formula, {' '})
    atomos = extraer_atomos(form)
    valoraciones = generar_valores(atomos)
    resultados = set()
    for asignacion in valoraciones:
        v.update(asignacion)
        resultado = valoracion(form, 0, len(form) - 1)
        if resultado == -1:
            return -1
        resultados.add(resultado)
    return 1 if len(resultados) == 1 and 1 in resultados else 0 if len(resultados) == 1 else -1
```

Una vez que se han generado todas las combinaciones posibles, el siguiente paso es evaluar la fórmula bajo cada una de estas asignaciones. Para esto, se recorre cada diccionario de valores de verdad y se actualiza el diccionario `v`, el cual ya fue utilizado en la Parte A para almacenar los valores de los átomos. Este diccionario es clave, ya que permite que la función `valoracion()`, implementada previamente, pueda evaluar la expresión correctamente al reemplazar cada átomo por su respectivo valor de verdad. Aquí es importante destacar que la función de valoración implementada en la Parte A es la que realmente se encarga de procesar la expresión de manera recursiva, identificando los operadores lógicos y aplicando las reglas de la lógica proposicional para determinar el valor de verdad de la fórmula completa.

Finalmente, la función principal `main_b()` es la que gestiona la interacción con el usuario.

```
def main_b():
    print("\nParte B\n")
    S = int(input())
    if 1 <= S <= 20:
        for l in range(S):
            formula = str(input())
            if len(formula) <= 32:
                print(evaluador(formula))
            else:
                print("Fórmula excede 32 caracteres")
    else:
        print("Fuera de rango (1 - 20)")

main_b()
```

En esta función, se solicita un número entero `S`, que indica cuántas fórmulas se van a evaluar. Se verifica que este número esté dentro del rango permitido (entre 1 y 20), y luego se permite al usuario ingresar las `S` fórmulas una por una. Cada una de estas fórmulas es evaluada con la función `evaluador()`, y se imprime el resultado correspondiente. Si alguna de las fórmulas ingresadas excede los 32 caracteres, el sistema lo indica y no la evalúa. Esta restricción se estableció para garantizar que las fórmulas sean manejables y no sobrepasen los límites establecidos en el problema. Así, se asegura que el usuario ingrese expresiones dentro de los parámetros esperados y que el programa pueda procesarlas sin inconvenientes.

```
1  Inizia a programmare o genera codice con l'IA.
```