Instituto Tecnológico y de Estudios Superiores de Occidente – ITESO

“FCC toolkit”
Equipo: Daner

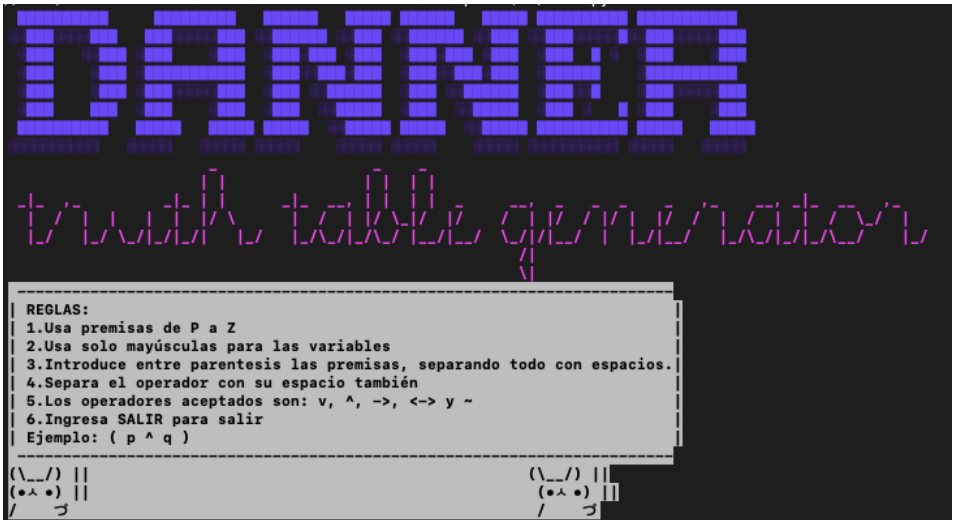
Materia:	***Fundamentos de Ciencias Computacionales***
Profesor:	Fernando Velasco Loera
Autora o autor(es):	Hernández Martínez Jennifer Ariadna Daniela Esparza Espinosa
Tema(s) de la tarea:	Generador de tablas de verdad
Fecha de entrega:	4 de marzo del 2021

Generador de tablas de verdad

Funcionamiento a nivel usuario final

1. Te aparecerá un banner con el nombre del equipo y de la herramienta
2. Podrás escribir de 1 a 11 premisas: P, Q, R, S, T, U, V, W, X, Y, Z
3. Las operaciones lógicas se representarán de la siguiente manera:
   - Conjunción “v”
   - Disyunción “^”
   - Condicional “->”
   - Bicondicional “<->”
   - Negación “~”
4. Las variables tendrán que escribirse con mayúsculas
5. Las premisas deberán de estar entre paréntesis y separadas con espacios
   Ejemplo: (P v Q)
6. Se desplazará la tabla de valores de las premisas ingresadas, después te dará la opción de seguir poniendo premisas hasta que escribas la palabra SALIR para poder terminar.

Ejemplos a probar
(P v Q) ^ R
(P v Q) -> R
(((P v Q) ^ R) -> S)



Documentación técnica

- Definimos las premisas que se podrán utilizar en la variable expressions, agregamos tanto las positivas como las negativas.
- Hicimos lo mismo con los operadores lógicos
- Agregamos un while para que las variables se iniciaran de nuevo y así poder generando tablas de verdad hasta que el usuario lo desee

```
v.proposition=""
while(v.proposition!="SALIR"):
    v.proposition=input(">>")
    v.arguments=v.proposition.split(' ')
    t.createTable(v.arguments)
    su.findParentheses(v.arguments)
    t.orderTable(v.tableValues)
    v.expressions={"P":[], "Q":[], "R":[], "S":[], "T":[], "U":[],
"V":[], "W":[], "X":[], "Y":[], "Z":[]
,"~P":[], "~Q":[], "~R":[], "~S":[], "~T":[], "~U":[], "~V":[], "~W":[],
"~X":[], "~Y":[], "~Z":[]}
    v.connectors={"^":0, "v":1, "->":2, "<->":3}
    v.variable=0
    v.variables={}
    v.values={}
    v.tableValues={}
    v.separator= "                                                    "
    v.rows=0
    v.truthTable={}
    v.finalsTable={}
    v.finalTable={}
    v.argument=""
```

- Regresa la posición donde encuentra por primera vez el carácter que recibe como parametro dentro de la proposición que tambien recibe.

```
def findFirst (proposition, letterToSearch): #Regresa la posición de la
primera ocurrencia de un caractér
    for i in range (len(proposition)): #Recorre la proposición
        if (proposition[i] == letterToSearch): #Si la encuentra
            return i #regresa la posición
    return -1 #regresa -1 si nunca aparece ese caracter en la proposición
#-------------------------------------------------------------------------
----------------------------
```

- Regresa la posición donde encuentra por última vez al carácter dentro de la proposición

```
- -

    def findLast (proposition, letterToSearch): #Regresa la posición
    de la última ocurrencia de un caracter
```

```
        for i in range (len(proposition)): #Recorre la proposición
            if (proposition[i]==letterToSearch): #Si lo encuentra
                if ((i+1) > len(proposition)): #Si era el último
    carácter de la proposición
                    return i #era la última ocurrencia
                search = findLast(proposition[i+1:len(proposition)],
    letterToSearch) #guardo el resultado de la busqueda
                if (search==-1): #si da menos uno quiere decir que
    ya no encontró otro después, el pasado era el último
                    return i #regreso el indice pasado
                else:
                    findLast(proposition[i+1:len(proposition)],
    letterToSearch) #si lo volvi a encontrar repito el proceso hasta
    si encontrar el ULTIMO
        return -1
```

-Encuentra los paréntesis dentro de la proposición que recibe como parametro, utiliza la variable auxiliar parenthesesUndone para asegurarse de rencontrar el paréntesis que cierra cada que encuentra uno que abre, a su vez extrae el contenido dentro de los parentesis y lo va guardando y evalua la expresión.

```
def findParentheses (proposition):
    openP=-1
    closeP=-1
    parenthesesUndone=0
    parenthesesCheck=0
    for i in range(len(proposition)):
        if (proposition[i]==')'): #Si encuentro un parentesis que cierra
resto uno al contador de pares porque está completo
            parenthesesUndone=parenthesesUndone-1
            closeP=i #Guardo la posicion donde cierro
        if (proposition[i]=='('): #Si encuentro un parentesis que abra
sumo uno al contador de pares incompletos porque todavía no encuentro su
cierre
            parenthesesUndone=parenthesesUndone+1
            openP=i #Guardo la posicion donde abro
        if (parenthesesUndone < parenthesesCheck):
            exp = proposition[openP+1:closeP] #extraes lo que está dentro
de los parentesis
            oldProp=saveOperation(exp)

            if (closeP== len(proposition)-1):
                newProposition = proposition[0:openP] + [oldProp]
            if (openP==0):
                newProposition = [oldProp] +
proposition[closeP+1:len(proposition)]
            if (openP!=0 and closeP != len(proposition)-1):
                newProposition =
proposition[0:openP]+[oldProp]+proposition[closeP:len(proposition)]
            if (findFirst(newProposition, '(')!=-1):
```

```
                findParentheses(newProposition)
            else:
                saveOperation(newProposition)
                op.evaluate()
                return
        parenthesesCheck=parenthesesUndone
```

- Quito los paréntesis de la expresión para no rebuscarlos y quedarme solo con variables y operadores, es una función recursiva donde se usa findFirst() y findLast() para encontrarlos, se extrae el fragmento y si aún hay paréntesis restantes se llama la función a si misma y repite el proceso.

```
def eliminateParentheses (proposition):
    openP=findLast(proposition, '(')
    newProposition=proposition[openP+1:len(proposition)]
    closeP=openP+findFirst(newProposition, ')')+1
    auxiliar=proposition[openP+1:closeP]
    var=saveOperation(auxiliar)

    argument=proposition[0:openP]
    argument.append(var)
    newProposition=proposition[closeP+1:len(proposition)]
    argument=argument+newProposition
    if findFirst(argument, '(') != -1:
        eliminateParentheses(argument)
    else:
        saveOperation(argument)
        op.evaluate()
```

- Las variables se renombran e inician con ‘v’, esta función restablece su nombre al final, evaluando la variable de nuevo

```
def renameExpressions(var,exp):
    if(len(v.variables)==1):
        op.evaluate(v.variables,var)
    for key, values in v.variables.items():
        for i in values:
            if i==var:
                pos=v.variables[key].index(i)
                v.truthTable[exp]=v.tableValues[exp]
```

```
                v.variables[key][pos]=exp
                op.evaluate(v.variables[key],key)
                return
#-------------------------------------------------------------------------
-----------------------------
```

- Esta función guarda la operación en el diccionario variables para su evaluación futura

```
def saveOperation(op):
    var='v'+str(v.variable)
    v.variables[var]=op
    v.variable=v.variable+1
    return var
```

- Las siguientes 4 funciones regresan el valor True o False dependiendo de la operación (conjunción, disyunción, implicación o bicondicional) a partir de los valores de sus tablas de verdad básicas

```
def logicAnd(arg1,arg2): #Regresa los valores de una operación Y a partir
de dos valores
    if (arg1==True) and (arg2==True): #Solo regresa verdad si ambos lo
son
        return True
    else: #Cualquier otra combinación regresa un falso
        return False
#-------------------------------------------------------------------------
-----------------------------
def logicOr(arg1,arg2): #Regresa los valores de una operación O a partir
de dos valores
    if (arg1==False) and (arg2==False): #Solo regresa falso si ambos lo
son
        return False
    else: #Cualquier otra combinación regresa verdadero
        return True
#-------------------------------------------------------------------------
-----------------------------
def logicConditional(arg1,arg2): #Regresa los valores de una operación
CONDICIONAL a partir de dos valores
    if (arg1==True)and(arg2==False): #Solo regresa Falso si el primero es
True y el segundo False
        return False
    else: #Cualquier otra combinación regresa verdadero
        return True
#-------------------------------------------------------------------------
-----------------------------
def logicBiconditional(arg1, arg2): #Regresa los valores de una operación
BICONDICIONAL a partir de dos valores
    if (arg1==True) and (arg2==False):
        return False
```

```
    elif (arg1==False) and (arg2==True):
        return False
    else: #Solo regresa verdadero si ambas son iguales
        return True
```

- Ejecuta la operación indicada por el operador entre los argumentos que reciba y lo guarda en la tabla. Recibe el operador, que se compara para saber a que función llamar y añade el resultado a la tabla.

```
def executeOperation(table, operator, arg1, arg2):
    newPos = arg1+operator+arg2
    table[newPos]=[]
    for i, j in zip(table[arg1],table[arg2]):
        result=""
        if operator=="^":
            result=logicAnd(i, j)
        if operator=="v":
            result=logicOr(i, j)
        if operator=="->":
            result=logicConditional(i, j)
        if operator=="<->":
            result=logicBiconditional(i, j)
        table[newPos].append(result)
    return [newPos, table]
```

- Evalúa las operaciones, necesita la ayuda de la ejecución de operación, marca el ritmo y hace que se operen todas las columnas como deben con recursividad.

```
def evaluate(exps=[], var=-1):
    if not exps:
        exps=v.variables['v'+str(v.variable-1)]
    argument2=""
    connector=""
    for exp in exps:
        if len(exp) == 2 and exp[0:1]=='v':
            newArgument=v.variables[exp]
            v.argument=''
            return evaluate(newArgument,exp)
        if (exp in v.expressions) or (exp in v.truthTable):
            v.tableValues[exp]=v.truthTable[exp]
            if v.argument == "":
                v.argument=exp
            elif exp!=v.argument:
                argument2=exp
                if v.argument==argument2:
                    v.argument=0
        if exp in v.connectors:
```

```
            connector=exp
        if (connector != "") and (v.argument!="") and (argument2!= ""):
            v.values=executeOperation(v.tableValues, connector,
v.argument, argument2)
            v.argument=v.values[0]
            v.tableValues=v.values[1]
            connector=""
            argument2=""
            su.renameExpressions(var,v.argument)
```

- Genera la tabla de acuerdo a las expresiones que reciba (Estas serán la proposición ingresada por el usuario dividida con Split en sus paréntesis).
- Primeramente, guarda las expresiones y la cantidad de ellas, calcula el número de filas utilizando la fórmula 2^n, siendo n la cantidad de premisas. Posteriormente revisa si la premisa es negativa o positiva y con un ciclo for añade sus respectivos True o False.

```
def createTable(expressions):
    exp={}
    expressionsQuantity=0
    for expression in expressions:
        if expression in v.expressions and not (expression in exp):
            expressionsQuantity=expressionsQuantity+1
            exp[expression]=''

    exp={}
    v.rows = 2**expressionsQuantity
    i= -1
    for expression in expressions:
        if expression in v.expressions and not (expression in exp):
            exp[expression]=''
            i=i+1
            notExp=False
            notExpression="~"+expression
            if(len(expression))==2:
                notExpression=expression[1:2]
                notExp=True
            v.truthTable[expression]=[]
            v.truthTable[notExpression]=[]
            divisor=2 ** (i+1)
            chunk=v.rows/divisor
            nextChunk=chunk
            addValue=True

            for j in range(v.rows):
                if j==nextChunk:
                    nextChunk=nextChunk+chunk
                    addValue=not addValue
                if notExp==True:
```

```
                    v.truthTable[notExpression].insert(j,addValue)
                    v.truthTable[expression].insert(j,not addValue)
                else:
                    v.truthTable[expression].insert(j,addValue)
                    v.truthTable[notExpression].insert(j,not addValue)
```

- Imprime la tabla con guiones y separadores para mayor legibilidad

```
def printTable(table):
    headers=""
    for i in table:
        headers=headers+"|     "+i+"    |"
    dividers=""
    for i in range(len(headers)):
        dividers=dividers+"-"
    print(Back.MAGENTA+Style.BRIGHT+Fore.WHITE+dividers)
    print(Back.MAGENTA+Style.BRIGHT+Fore.WHITE+headers)
    print(Back.MAGENTA+Style.BRIGHT+Fore.WHITE+dividers)

    printing=""
    for i in range(v.rows):
        for j,k in table.items():
            if k[i]==True:
                printing=printing+"| True"+v.separator[0:len(j)+5-1]+"|"
            else:
                printing=printing+"| False"+v.separator[0:len(j)+4-1]+"|"
        print(printing)
        printing=""
    print(Back.MAGENTA+Style.BRIGHT+Fore.WHITE+dividers,"\n")
```

- Ordena la tabla poniendo primero variables positivas, luego variables negativas, finalmente los argumentos con operacion

```
def orderTable(table):
    positiveTable={}
    negativeTable={}
    operationsTable={}
    finalTable={}
    for exp in table.keys():
        if exp in v.expressions:
            if len(exp)==2:
                contrary=exp[1:2]
                negativeTable[exp]=v.truthTable[exp]
                positiveTable[exp]=v.truthTable[contrary]
            else:
                positiveTable[exp]=v.truthTable[exp]
        else:
            operationsTable[exp]=table[exp]
    for key, value in positiveTable.items():
        finalTable[key]=value
    for key, value in negativeTable.items():
        finalTable[key]=value
```

```
    for key, value in operationsTable.items():
        finalTable[key]=value
    printTable(finalTable)
```