

# Índice general

<b>I</b>	<b>Aparato lógico</b>	<b>2</b>
1.	Estructura del paquete <code>calcprop</code>	3
2.	Clase <code>Proposicion</code> y su subclase <code>v</code> ( <b>proposición atómica</b> )	4
2.1.	La subclase <code>v</code> para proposiciones atómicas (i.e., para variables proposicionales)	5
3.	<b>Valoración de proposiciones</b>	<b>6</b>
3.1.	Función <code>span</code>	7
3.1.1.	Implementación	7
4.	<b>Función <code>extension</code> que extiende una función parcial</b>	<b>8</b>
4.1.	Implementación de la función <code>extension</code>	8
4.1.1.	Implementación	9
4.2.	Reducción de proposiciones NO atómicas	9
4.3.	Proposición no derivable de un conjunto de <b>premisas</b>	10
4.4.	... cuando llegamos a una proposición atómica (o <i>variable proposicional</i> )	11
5.	<b>Función que refuta una proposición a partir de un conjunto de premisas</b>	<b>12</b>
6.	<b>Función <code>test</code></b>	<b>13</b>
7.	<b>Representación de la clase <code>Proposicion</code></b>	<b>14</b>

Parte I

Aparato lógico

# Capítulo 1

## Estructura del paquete calcprop

---

```
calcprop/__init__.py
```

---

```
<<Aparato lógico de la librería>>
```

---

```
Aparato lógico de la librería
```

---

```
<<Definición de la clase Proposicion>>  
<<Definición de la subclase v>>  
<<Definición de alguno y unoDe>>  
<<Definición de noDerivable>>  
<<Definición de la función span>>  
<<Definición de la función extension>>  
<<Definición de la función refuta>>  
<<Definición de la función test>>
```

---

El <<Aparato auxiliar para la creación de problemas de opción múltiple>> se desarrolla en el paquete calcprop-quiz.

## Capítulo 2

# Clase Proposicion y su subclase v (proposición atómica)

Las proposiciones son expresiones formadas con números, strings y los símbolos

`&`, `|`, `-`, `v( )`, `>>`, `**`, `alguno` y `unoDe`

construidas según las siguientes reglas:

1. Si  $D$  es un número o un string (cadena de caracteres) entonces  $v(D)$  es una subclase de `Proposicion` que denominaremos “*variable proposicional*” o “*proposición atómica*”.
2. Si  $P$ ,  $Q$  y  $R$  son proposiciones también lo son:

$P \& Q$  (conjunción)

$P | Q$  (disyunción)

$\neg P$  (negación)

$P \gg Q$  (implicación)

$P ** Q$  (equivalencia, es decir: *si y solo si*).

Para facilitar la elaboración expresiones, también incluimos dos operadores lógico menos habituales:

`alguno(P,Q,R)` significa que alguna de las proposiciones de la lista es verdadera y

`unoDe(P,Q,R)` significa que solo una de las proposiciones de la lista es verdadera.

Ejemplo de proposición: “llueve o no llueve”

---

```
v('llueve') | -v('llueve')
```

---

```
v('llueve') | -v('llueve')
```

Necesitamos que las proposiciones del tipo  $v(D)$  sean “*hash-eables*”, para poder ser utilizadas como claves en un diccionario. Para ello necesitamos los métodos `__hash__` y `__eq__`.

Definamos la clase `Proposicion`

---

Definición de la clase `Proposicion`

---

```
class Proposicion:
    def __init__(self,data):
        self.data = data

    def __hash__(self):
        return hash(self.data)

    def __eq__(self, another):
        return hasattr(another, 'data') and self.data == another.data
```

<<Operadores lógicos de la clase `Proposicion`>>

<<Método de representación de la clase `Proposicion`>>

---

1. Las operaciones: conjunción, disyunción, negación, implicación y equivalencia.

Definamos como métodos dentro de la clase `Proposicion` los cinco operadores lógicos habituales. Primero aquellos que usan “*métodos mágicos*”, es decir, aquellos que tiene asociado un símbolo para llamar a las operaciones:

---

Operadores lógicos de la clase `Proposicion`

```
def __and__(self,other):
    return Proposicion(['and', self, other])
def __or__(self,other):
    return Proposicion(['or', self, other])
def __neg__(self):
    return Proposicion(['not', self])
def __rshift__(self,other):
    return Proposicion(['implica', self, other])
def __pow__(self,other):
    return Proposicion(['equivale', self, other])
```

---

Las operaciones: `unoDe` y `alguno`.

Para las dos siguientes operaciones<sup>1</sup> no usamos ningún “*método mágico*”. Así que definir las dentro de la clase nos obligaría a una incómoda escritura del tipo: `Proposicion.alguno(P1,P2,P3)` o `Proposicion.unoDe(P1,P2,P3)`. Para evitarlo, las definimos fuera de la clase; así podremos escribir sencillamente `alguno(P1,P2,P3)` y `unoDe(P1,P2,P3)`.

---

Definición de `alguno` y `unoDe`

```
def alguno(X,*args):
    return Proposicion(['alguno', X] + [a for a in args])
def unoDe(X,*args):
    return Proposicion(['unoDe', X] + [a for a in args])
```

---

La operación: `noDerivable`.

Esta función no es una operación lógica (como si lo son las anteriores). Es un “añadido” cuya motivación es la siguiente:

El esquema típico de una pregunta es: el enunciado asume que las premisas  $E_1, E_2, \dots$  son ciertas. Y se puede preguntar si  $P$  es consecuencia lógica de las premisas. Si lo es,  $P$  es `True` (es decir,  $\text{premisas} \Rightarrow P$ ). Negar esto no es decir que  $\neg P$  es consecuencia de las premisas, sino que  $P$  no es derivable de las premisas (es decir,  $\text{premisas} \not\Rightarrow P$ ). Así que añadimos la operación `noDerivable`. Esto nos permite, por ejemplo, que aunque para algunos enunciados se pueda preguntar si una matriz es diagonalizable, para otros evitemos la pregunta si no es deducible del enunciado.

---

Definición de `noDerivable`

```
def noDerivable(X):
    return Proposicion(['noDerivable', X])
```

---

## 2.1. La subclase `v` para proposiciones atómicas (i.e., para variables proposicionales)

---

Definición de la subclase `v`

```
class v(Proposicion):
    def __init__(self, data):
        super().__init__(data)
```

---

---

<sup>1</sup>no habituales, aunque convenientes pues facilitan la expresión de algunas proposiciones

## Capítulo 3

# Valoración de proposiciones

Una valoración es una función  $T$  que va del conjunto de proposiciones al conjunto  $\{\text{True}, \text{False}\}$  (es decir, a cada proposición le asigna el valor verdadero o el valor falso) y que verifica las siguientes propiedades: si  $P$  y  $Q$  son proposiciones entonces

1.  $T(P \ \& \ Q) = \text{True}$  únicamente en el caso de que  $T(P) = \text{True}$  y  $T(Q) = \text{True}$ .
2.  $T(P \ | \ Q) = \text{False}$  únicamente en el caso de que  $T(P) = \text{False}$  y  $T(Q) = \text{False}$ .
3.  $T(\neg P) = \text{True}$  únicamente en el caso de que  $T(P) = \text{False}$ .

Por tanto:  $T(P \ \& \ Q) = T(P) \ \& \ T(Q)$ ,  $T(P \ | \ Q) = T(P) \ | \ T(Q)$  y  $T(\neg P) = \text{not } T(P)$ .

1.  $T(P \ \gg \ Q) = \text{False}$  únicamente en el caso en que  $T(P) = \text{True}$  y  $T(Q) = \text{False}$ .
2.  $T(P \ \ast\ast \ Q) = \text{True}$  únicamente en el caso en que  $T(P) = T(Q)$ .

Por tanto

$$T(P \ Q) = T(\neg P \ | \ Q) \\ \text{y } P)$$

$$T(P \ \ast\ast \ Q) = T((P \ Q) \ \& \ (Q$$

Y si  $A_1, \dots, A_n$  son  $n$  proposiciones, entonces

1.  $\text{alguno}(A_1, \dots, A_n)$  es  $\text{True}$  si y solo si  $A_1 \ | \ \text{alguno}(A_2, \dots, A_n)$  es  $\text{True}$ ,
2.  $\text{unoDe}(A_1, \dots, A_n)$  es  $\text{True}$  si y solo si  $(A_1 \ \& \ \neg \text{alguno}(A_2, \dots, A_n) \ | \ (\neg A_1 \ \& \ \text{unoDe}(A_2, \dots, A_n)))$  es  $\text{True}$ ;

Por tanto

$$T(\text{alguno}(A_1, \dots, A_n)) = T(A_1 \ | \ \text{alguno}(A_2, \dots, A_n))$$

y

$$T(\text{unoDe}(A_1, \dots, A_n)) = T((A_1 \ \& \ \neg \text{alguno}(A_2, \dots, A_n)) \ | \ (\neg A_1 \ \& \ \text{unoDe}(A_2, \dots, A_n)))$$

Cualquier función  $F$  que vaya del conjunto de proposicionales atómicas (es decir, del tipo  $v(\text{'algo'})$ ) al conjunto  $\{\text{True}, \text{False}\}$  se puede extender al resto proposiciones<sup>1</sup> formando una valoración  $T = \text{span}(F)$ , aplicando las reglas de arriba. Dicha extensión es única.

Vamos a implementar como funciones los diccionarios de Python, pues los diccionarios son una colección de pares  $\text{'Etiqueta': valor}$  donde a cada etiqueta solo se le asigna un valor único. Por ejemplo, si definimos el siguiente diccionario  $\{\text{'a':1}, \text{'a':0}, \text{'b':2}\}$ , obtenemos un diccionario con dos elementos, donde a  $\text{'a'}$  se le ha asignado solo uno de los dos valores introducidos (en concreto el último).

```
print( {'a':1, 'a':0, 'b':2} )
```

```
{'a': 0, 'b': 2}
```

<sup>1</sup>En la función `span` de Python hemos elegido el criterio de que a toda proposición `Prop` no incluida explícitamente en el diccionario `F` le asignamos el valor `False` para así completar el dominio de las proposiciones atómicas. Esta decisión ha sido completamente arbitraria, y se ha incluido para que efectivamente el dominio de `span` incluya todas las proposiciones atómicas (i.e., variables proposicionales). En cualquier caso `span` se ha programado a modo de ilustración, pues no la emplearemos para nada.

## 3.1. Función span

Función `span(F, Prop)` indica si `Prop` es `True` o es `False` a partir de la función parcial `F`.

- `F`: es un diccionario (función parcial) que asigna valores `True` o `False` a varias proposiciones atómicas.
- `Prop`: es una `Proposicion` .

Así obtenemos los siguientes resultados lógicos:

---

```
F = { v('llueve'):True }
span(F, v('llueve') >> ~v('llueve') )
```

---

False

y por otra parte

---

```
span(F, ~v('llueve') >> v('llueve') )
```

---

True

### 3.1.1. Implementación

Si una proposición `Prop` (de tipo atómico) está en la función parcial `F` y además es verdadera, `span` devuelve `True`. En caso contrario devuelve `False` (tanto si `Prop` no está en `F` como si lo está pero es `False`). Si `Prop` no es de tipo atómico, entonces se aplican las reglas para reducir `Prop` a proposiciones de tamaño más pequeño.

---

Definición de la función span

```
def span(F, Prop):
    if type(Prop) == type(v('')):
        return (Prop in F) and F[Prop]
    <<Aplicación de las reglas para reducir el tamaño de Prop>>
```

---

Aplicación de las reglas para reducir el tamaño de Prop

```
elif Prop.data[0] == 'and':
    return span(F, Prop.data[1]) and span(F, Prop.data[2])

elif Prop.data[0] == 'or':
    return span(F, Prop.data[1]) or span(F, Prop.data[2])

elif Prop.data[0] == 'not':
    return not span(F, Prop.data[1])

elif Prop.data[0] == 'implica':
    return (not span(F, Prop.data[1])) or span(F, Prop.data[2])

elif Prop.data[0] == 'equivale':
    return span(F, Prop.data[1]) == span(F, Prop.data[2])

elif Prop.data[0] == 'alguno':
    return span(F, Prop.data[1]) or span(F, alguno(Prop.data[2:])) if len(Prop.data)>2\
    else span(F, Prop.data[1])

elif Prop.data[0] == 'unoDe':
    A = Prop.data[1]
    if len(Prop.data)==2:
        return span(F, A)
    else:
        B = Proposicion(['alguno'] + Prop.data[2:])
        C = Proposicion(['unoDe'] + Prop.data[2:])
        return span(F, (A&~B) | (~A&C) )
```

---

## Capítulo 4

# Función `extension` que extiende una función parcial

Ahora queremos definir una función de Python (que denominamos `extension`) que, dadas una lista `objetivo` y un diccionario con `valores`, para una colección de proposiciones atómicas haga crecer (extienda) el diccionario `valores` con nuevas proposiciones atómicas, y sus correspondientes valores, necesarios para que se cumplan los pares contenidos en la lista `objetivo`; o bien que devuelva el diccionario vacío `{}` en caso de que sea imposible hacer cumplir dicha lista `objetivo`.

La diferencia entre la lista `objetivo` y los `valores` es que en `objetivo` se asignan valores a proposiciones (arbitrarias) pero en `valores` solo se asignan valores a *proposiciones atómicas*.

El diccionario extendido es un *modelo lógico* en el que se cumplen la lista `objetivo` y la función parcial `valores`. Ello no quiere decir que no puedan existir otros modelos que consigan lo mismo.

Ejemplo:

---

```
print( extension( [ (v('llueve') >> ~v('llueve')) , True ] , {v('basilisco') : True} ) )
```

---

```
{v('basilisco'): True, v('llueve'): False}
```

Ejemplo:

---

```
print( extension( [ (v('llueve'),True), (~v('llueve'),True)], {v('basilisco') : True} ) )
```

---

```
{}
```

### 4.1. Implementación de la función `extension`

Función `extension(objetivo, valores, premisas=[])`

- `objetivo`: es una lista de pares  $(P, VF)$  donde  $P$  es una proposición y  $VF$  es `True` o `False`
- `valores`: es un diccionario (función parcial) que asigna valores `True` o `False` a variables proposicionales
- `premisas`: es la lista de premisas a partir de la cual se deduce si una proposición es derivable o no (es decir, es solo para usar la operación `noDerivable`)

La función `extension` tiene dos posibles salidas: o bien devuelve un modelo lógico  $M$  que extiende a `valores` de tal forma que  $\text{span}(M, P) = VF$  para cada par  $(P, VF)$  de `objetivo`, o el diccionario vacío `{}` si no existe tal modelo  $M$ . Es decir, busca una forma de dar valores a las proposiciones atómicas de modo que todas las proposiciones contenidas en el `objetivo` alcancen el valor deseado.

### 4.1.1. Implementación

Si la lista `objetivo` no tiene pares, con la propia función `valores`, hemos acabado.

La definición de la función `extension` es recursiva. La mecánica es la siguiente: se sustituye el proposición del primer objetivo por proposiciones de menor tamaño<sup>1</sup>, y entonces se llama a si misma. Este procedimiento reduce las proposiciones hasta obtener proposiciones atómicas.

Así pues, llamamos `Obj` al primer objetivo de la lista, `P` a la correspondiente proposición y `VF` a su valor objetivo. Por tanto `Obj = (P,VF)` es el primer par de la lista `objetivo`.

Comprobamos que `P.data` es una `list` y tratamos los distintos casos de reducción de predicado.

---

Definición de la función `extension`

```
def extension(objetivo, valores, premisas=[]):
    if objetivo == []:
        return valores

    Obj = objetivo[0]
    P = Obj[0]
    VF = Obj[1]
    if isinstance(P.data,list):
        <<Reducción de una proposición de tipo "not">>
        <<Reducción de una proposición de tipo "and">>
        <<Reducción de una proposición de tipo "or">>
        <<Reducción de una proposición de tipo "implica">>
        <<Reducción de una proposición de tipo "equivale">>
        <<Reducción de una proposición de tipo "alguno">>
        <<Reducción de una proposición de tipo "unoDe">>
        <<Proposición de tipo "noDerivable">>
    <<Proposición atómica v>>
```

---

## 4.2. Reducción de proposiciones NO atómicas

### 1. Reducción de una proposición de tipo “not”.

En el caso de que `Obj = (-A, True)`, nos vale la misma función que obtenemos si reemplazamos `Obj` por `(A, False)`. Y en el caso de que `Obj = (-A, False)`, nos vale la misma función si reemplazamos `Obj` por `(A, True)`.

---

Reducción de una proposición de tipo "not"

```
if P.data[0] == 'not' :
    A = P.data[1]
    return extension( [(A, not VF)] + objetivo[1:], valores, premisas )
```

---

### 2. Reducción de una proposición de tipo “and”.

En el caso de que `Obj=(A & B, True)`, nos vale la misma función que obtenemos si reemplazamos `Obj` por `(A, True)`, `(B, True)`. Y en el caso de que `Obj=(A & B, False)`, vale tanto la función que obtenemos de reemplazar `Obj` por `(A, False)` como la que obtenemos al reemplazar `Obj` por `(B, False)`.

Si la primera alternativa resulta en una extensión vacía se devuelve la segunda.

---

Reducción de una proposición de tipo "and"

```
elif P.data[0] == 'and':
    A = P.data[1]
    B = P.data[2]
    if(VF):
        return extension([(A, True), (B, True)] + objetivo[1:], valores, premisas)
    else:
        r = extension([(A, False)] + objetivo[1:], valores, premisas)
        return extension([(B, False)] + objetivo[1:], valores, premisas) if not r else r
```

---

### 3. Reducción de una proposición de tipo “or”.

En el caso de que `Obj=( A|B, False)`, nos vale la misma función que obtenemos si reemplazamos `Obj` por `(A, False)`, `(B, False)`. Y en el caso de que `Obj=( A&B, True)`, vale tanto la función que obtenemos de reemplazar `Obj` por `(A, True)` como la que obtenemos al reemplazar `Obj` por `(B, True)`.

Si la primera alternativa resulta en una extensión vacía se devuelve la segunda.

---

<sup>1</sup>Una lista de símbolos más corta

---

```

Reducción de una proposición de tipo "or"
elif P.data[0] == "or":
    A = P.data[1]
    B = P.data[2]
    if not VF:
        return extension([(A,False),(B,False)] + objetivo[1:], valores, premisas)
    else:
        r = extension([(A,True)] + objetivo[1:], valores, premisas)
        return extension([(B,True)]+objetivo[1:],valores, premisas) if not r else r

```

---

#### 4. Reducción de una proposición de tipo "implica".

En el caso de que  $Obj = (A \gg B, VF)$ , nos vale la misma función que obtenemos si emplazamos  $Obj$  por  $(\neg A \vee B, VF)$ .

---

```

Reducción de una proposición de tipo implica"
elif P.data[0] == "implica":
    A = P.data[1]
    B = P.data[2]
    return extension([(¬A|B, VF)] + objetivo[1:], valores, premisas)

```

---

#### 5. Reducción de una proposición de tipo "equivale".

En el caso de que  $Obj = (A \leftrightarrow B, VF)$ , nos vale la misma función que obtenemos si remplazamos  $Obj$  por  $((A \gg B) \& (B \gg A), VF)$ .

---

```

Reducción de una proposición de tipo "equivale"
elif P.data[0] == "equivale":
    A = P.data[1]
    B = P.data[2]
    return extension([(A>>B) & (B>>A), VF]) + objetivo[1:], valores, premisas)

```

---

#### 6. Reducción de una proposición de tipo "alguno".

En caso de que  $Obj = (\text{alguno}(A), VF)$  (i.e., la lista de alternativas solo contiene la proposición  $A$ ) nos vale la misma función que  $(A, VF)$ . Si por el contrario  $Obj = (\text{alguno}(A_1, \dots, A_n), VF)$ , con  $n > 1$ , nos vale la misma función que obtenemos si remplazamos  $Obj$  por  $(A_1 \vee \text{alguno}(A_2, \dots, A_n), VF)$  (i.e., la primera o alguna de las demás).

---

```

Reducción de una proposición de tipo "alguno"
elif P.data[0] == "alguno":
    A = P.data[1]
    if len(P.data)==2:
        return extension([(A,VF)] + objetivo[1:], valores, premisas)
    else:
        B = Proposicion(["alguno"] + P.data[2:])
        return extension([(A|B, VF)] + objetivo[1:], valores, premisas)

```

---

#### 7. Reducción de una proposición de tipo "unoDe".

En caso de que  $Obj = (\text{unoDe}(A), VF)$  (i.e., la lista de alternativas solo contiene la proposición  $A$ ) nos vale la misma función que  $(A, VF)$ . Si por el contrario  $Obj = (\text{unoDe}(A_1, \dots, A_n), VF)$ , con  $n > 1$ , nos vale la misma función que obtenemos si remplazamos  $Obj$  por  $(A_1 \& \neg \text{alguno}(A_2, \dots, A_n) \vee \neg A_1 \& \text{unoDe}(A_2, \dots, A_n), VF)$ .

---

```

Reducción de una proposición de tipo unoDe"
elif P.data[0] == "unoDe":
    A = P.data[1]
    if len(P.data)==2:
        return extension([(A,VF)] + objetivo[1:], valores, premisas)
    else:
        B = Proposicion(["alguno"] + P.data[2:])
        C = Proposicion(["unoDe"] + P.data[2:])
        return extension([(A&¬B) | (¬A&C), VF]) + objetivo[1:], valores, premisas)

```

---

### 4.3. Proposición no derivable de un conjunto de premisas

La función `extension` tiene un tercer argumento (`premisas`) que es un "añadido" para decidir si una proposición es noDerivable de la lista de premisas.

Derivable significa que la proposición es consecuencia lógica de las premisas, por consiguiente es imposible que sea falso si las premisas son ciertas.

---

```

elif P.data[0] == "noDerivable":
    A = P.data[1]
    valoresExt = valores.copy()
    valoresExt[v(repr(P))] = VF

    derivable = not extension([(A,False)] + premisas, {}, premisas)

    return extension(objetivo[1:],valores2,premisas) if not derivable == VF else {}

```

---

#### 4.4. ... cuando llegamos a una proposición atómica (o *variable proposicional*)

En el caso de que  $\text{Obj} = (v(D), VF)$ , es cuando el algoritmo recursivo pasará a intentar un nuevo objetivo (extendiendo el diccionario `valores` si fuera necesario), o bien parará por ser el par objetivo  $(v(D), VF)$  imposible.

Primero se comprueba si el par  $(v(D), VF)$  ya estaba en el diccionario `valores`, es decir, si `valores` contiene la proposición atómica  $v(D)$  y con el valor `VF`. En tal caso, se “salta” al siguiente objetivo (es decir se llama al `extension` con el mismo diccionario `valores` pero quitando de la lista objetivo el primer elemento. Si  $v(D)$  pertenece al diccionario `valores` pero **con un valor distinto** a `VF`, quiere decir que el objetivo es imposible de alcanzar, por lo que se devuelve un diccionario vacío, y la recursión ha terminado.

Si  $P$  es una proposición atómica que no estaba en el diccionario `valores`, entonces creamos un nuevo diccionario extendido (`valoresExt`) que incluya  $P$  y su valor `VF`; y entonces llamamos a `extension` pero quitando el primer objetivo de la lista y usando un diccionario “extendido” (`valoresExt`). Para evitar los efectos colaterales de modificar un diccionario, modificamos una copia

---

```

else:
    if P in valores:
        return extension(objetivo[1:], valores, premisas) if valores[P]==VF else {}
    else:
        valoresExt = valores.copy()
        valoresExt[P] = VF
        return extension(objetivo[1:], valoresExt, premisas)

```

---

## Capítulo 5

# Función que refuta una proposición a partir de un conjunto de premisas

Refutar una proposición, consiste en asignar un valor a cada variable proposicional (cada proposición atómica), de tal modo que el resultado sea falso.

La función `refuta` construye un modelo  $M$  (asignación de valores a las proposiciones atómicas) de forma que  $\text{span}(M,P)=\text{False}$  y  $\text{span}(M,h)=\text{True}$  para todo  $h$  en `premisas`.

Si esto no es posible refutar  $P$  a partir de las `premisas`, entonces la función `refuta` devuelve un modelo (un diccionario) vacío; esto quiere decir que la proposición  $P$  es irrefutable, es decir,  $\text{premisas} \Rightarrow P$ . Es decir, si se refuta es que no es derivable de las premisas (no es consecuencia lógica de las premisas).

---

Definición de la función `refuta`

---

```
def refuta(P, premisas=[]):  
    h=[(Q,True) for Q in premisas]  
    return extension([(P,False)]+h, {}, h)
```

---

# Capítulo 6

## Función test

Función `test(P,premisas)`

- `P`: es una proposición o cualquiera de los valores `True`, `False`.
- `premisas`: es una lista de proposiciones

La función `test` se utiliza tanto para saber si se da una precondición, como para saber el resultado correcto de una cuestión.

En el caso de que `P` sea una proposición, sólo devuelve `True` si es imposible refutarlo a partir de las `premisas`, es decir, es imposible que `P` sea falso si las `premisas` son verdad (`premisas  $\Rightarrow$  P`, i.e., `P` es derivable).

---

Definición de la función test

---

```
def test(P,premisas=[]):
    if P == True:
        return True
    if P == False:
        return False
    else:
        return True if not refuta(P, premisas) else False
```

---

## Capítulo 7

# Representación de la clase Proposicion

---

Método de representación de la clase Proposicion

```
def __repr__(self):
    if isinstance(self.data, list):
        if self.data[0] == "and":
            return repr(self.data[1]) + " & " + repr(self.data[2])
        if self.data[0] == "or":
            return repr(self.data[1]) + " | " + repr(self.data[2])
        if self.data[0] == "not":
            return "-" + repr(self.data[1])
        if self.data[0] == "implica":
            return repr(self.data[1]) + " >> " + repr(self.data[2])
        if self.data[0] == "equivale":
            return repr(self.data[1]) + " ** " + repr(self.data[2])
        if self.data[0] == "unoDe":
            return "unoDe('+', '.join([repr(x) for x in self.data[1:]] +)")"
        if self.data[0] == "noDerivable":
            return "noDerivable("+repr(self.data[1]) +")"
    else:
        return 'v('+repr(self.data)+')
```

---