Práctica Nº 7 - Resolución en Lógica

Resolución en Lógica Proposicional

Ejercicio 1 ★

Convertir a Forma Normal Conjuntiva y luego a Forma Clausal (notación de conjuntos) las siguientes fórmulas proposicionales:

I. $P \Rightarrow P$

II. $(P \wedge Q) \Rightarrow P$

III. $(P \vee Q) \Rightarrow P$

IV. $\neg(P \Leftrightarrow \neg P)$

V. $\neg(P \wedge Q) \Rightarrow (\neg P \vee \neg Q)$

VI. $(P \wedge Q) \vee (P \wedge R)$

VII. $(P \wedge Q) \Rightarrow R$

VIII. $P \Rightarrow (Q \Rightarrow R)$

Ejercicio 2 ★

I. ¿Cuáles de las fórmulas del ejercicio anterior son tautologías? Demostrarlas utilizando el método de resolución para la lógica proposicional. Para las demás, indicar qué pasa si se intenta demostrarlas usando este método.

II. ¿Se deduce $(P \wedge Q)$ de $(\neg P \Rightarrow Q) \wedge (P \Rightarrow Q) \wedge (\neg P \Rightarrow \neg Q)$? Contestar utilizando el método de resolución para la lógica proposicional.

Ejercicio 3

Demostrar las siguientes tautologías utilizando el método de resolución para la lógica proposicional. Notar que no siempre es necesario usar todas las cláusulas.

- $(P \Rightarrow (P \Rightarrow Q)) \Rightarrow (P \Rightarrow Q)$
- $(R \Rightarrow \neg Q) \Rightarrow ((R \wedge Q) \Rightarrow P)$
- $((P \Rightarrow Q) \Rightarrow (R \Rightarrow \neg Q)) \Rightarrow \neg(R \wedge Q)$

Ejercicio 4 ★

Un grupo de amigos quería juntarse a comer en una casa, pero no decidían en cuál. Prevalecían dos propuestas: la casa de Ana, que era cómoda y espaciosa, y la de Carlos, más chica pero con un amplio jardín y parrilla al aire libre. Finalmente acordaron basar su elección en el pronóstico del tiempo. Si anunciaban lluvia, se reunirían en la casa de Ana; y si no, en la de Carlos (desde ya, la reunión tendría lugar en una sola casa).

Finalmente llego el día de la reunión, y el grupo se juntó a comer en la casa de Ana, pero no llovió.

Utilizar las siguientes proposiciones para demostrar - mediante el método de resolución - que el pronóstico se equivocó (anunció lluvia y no llovió, o viceversa).

P = "El pronóstico anunció lluvia."

A = "El grupo se reúne en la casa de Ana."

C = "El grupo se reúne en la casa de Carlos."

L = "Llueve en el día de la reunión."

Ayuda: por la descripción de arriba sabemos que $P \Rightarrow A$, $\neg P \Rightarrow C$ y $\neg(A \wedge C)$, además de que A y $\neg L$ son verdaderas. Pensar en lo que se quiere demostrar para decidir qué pares de cláusulas utilizar.

Resolución en Lógica de Primer Orden

En esta sección, salvo que se haga referencia a SLD, la palabra *resolución* denota el método de resolución general. Siempre que se demuestre una propiedad, se deberá indicar la sustitución utilizada en cada paso de resolución.

Ejercicio 5

Convertir a Forma Normal Negada (NNF) las siguientes fórmulas de primer orden:

I. $\forall X.\forall Y.(\neg Q(X,Y) \Rightarrow \neg P(X,Y))$

II. $\forall X.\forall Y.((P(X,Y) \wedge Q(X,Y)) \Rightarrow R(X,Y))$

III. $\forall X.\exists Y.(P(X,Y) \Rightarrow Q(X,Y))$

Ejercicio 6 ★

Convertir a Forma Normal de Skolem y luego a Forma Clausal las siguientes fórmulas de primer orden:

I. $\exists X.\exists Y.X < Y$, siendo $<$ un predicado binario usado de forma infija.

II. $\forall X.\exists Y.X < Y$

III. $\forall X.\neg(P(X) \wedge \forall Y.(\neg P(Y) \vee Q(Y)))$

IV. $\exists X.\forall Y.(P(X,Y) \wedge Q(X) \wedge \neg R(Y))$

V. $\forall X.(P(X) \wedge \exists Y.(Q(Y) \vee \forall Z.\exists W.(P(Z) \wedge \neg Q(W))))$

Ejercicio 7

Para pensar (o jugar):

I. Exhibir una cláusula que arroje un resolvente consigo misma.

II. Exhibir dos cláusulas, cada una con no más de dos literales, que arrojen tres o más resolventes distintos entre sí.

III. Exhibir dos cláusulas que arrojen como resolvente $\square$ si se unifican tres o más términos a la vez, pero no si se unifica solamente un término de cada lado.

Ejercicio 8 ★

La computadora de la policía registró que el Sr. Smullyan no pagó una multa. Cuando el Sr. Smullyan pagó la multa, la computadora grabó este hecho pero, como el programa tenía errores, no borró el hecho que expresaba que no había pagado la multa. A partir de la información almacenada en la computadora, mostrar utilizando resolución que el jefe de gobierno es un espía.

Utilizar los siguientes predicados y constantes: *Pagó*(X) para expresar que X pagó su multa, *Espía*(X) para X es un espía, smullyan para el Sr. Smullyan y jefeGob para el jefe de gobierno.

Ejercicio 9 ★

¿Cuáles de las siguientes fórmulas son lógicamente válidas? Demostrar las que lo sean usando resolución.

I. $[\exists X.\forall Y.R(X,Y)] \Rightarrow \forall Y.\exists X.R(X,Y)$

II. $[\forall X.\exists Y.R(X,Y)] \Rightarrow \exists Y.\forall X.R(X,Y)$

III. $\exists X.[P(X) \Rightarrow \forall X.P(X)]$

IV. $\exists X.[P(X) \vee Q(X)] \Rightarrow [\exists X.P(X) \vee \exists X.Q(X)]$

V. $\forall X.[P(X) \vee Q(X)] \Rightarrow [\forall X.P(X) \vee \forall X.Q(X)]$

VI. $[\exists X.P(X) \wedge \forall X.Q(X)] \Rightarrow \exists X.[P(X) \wedge Q(X)]$

VII. $\forall X.\exists Y.\forall Z.\exists W.[P(X,Y) \vee \neg P(W,Z)]$

VIII. $\forall X.\forall Y.\forall Z.([\neg P(f(\mathsf{a})) \vee \neg P(Y) \vee Q(Y)] \wedge P(f(Z)) \wedge [\neg P(f(f(X))) \vee \neg Q(f(X))])$

Ejercicio 10 *(Aplicaciones del método de resolución)*

I. Expresar en forma clausal la regla del *modus ponens* y mostrar que es válida, usando resolución.

II. Lo mismo para la regla del *modus tollens*.

III. Lo mismo para la regla de especialización: de $\forall X P(X)$ concluir $P(t)$ cualquiera sea el término t.

Ejercicio 11 ★

Dadas las siguientes cláusulas:

- $\{P(X), \neg P(X), Q(\mathsf{a})\}$
- $\{P(X), \neg Q(Y), \neg R(X,Y)\}$
- $\{\neg P(X,X,Z), \neg Q(X,Y), \neg Q(Y,Z)\}$
- $\{M(1,2,X)\}$

I. ¿Cuáles son cláusulas de Horn?

II. Para cada cláusula de Horn indicar si es una cláusula de definición (hecho o regla) o una cláusula objetivo.

III. Dar, para cada cláusula, la fórmula de primer orden que le corresponde.

Ejercicio 12 ★

Indicar cuáles de las siguientes condiciones son necesarias para que una demostración por resolución sea SLD.

- Realizarse de manera lineal (utilizando en cada paso el resolvente obtenido en el paso anterior).
- Utilizar únicamente cláusulas de Horn.
- Utilizar cada cláusula a lo sumo una vez.
- Empezar por una cláusula objetivo (sin literales positivos).
- Empezar por una cláusula que provenga de la negación de lo que se quiere demostrar.
- Recorrer el espacio de búsqueda de arriba hacia abajo y de izquierda a derecha.
- Utilizar la regla de resolución binaria en lugar de la general.

Ejercicio 13 ★

Alan es un robot japonés. Cualquier robot que puede resolver un problema lógico es inteligente. Todos los robots japoneses pueden resolver todos los problemas de esta práctica. Todos los problemas de esta práctica son lógicos. Existe al menos un problema en esta práctica. ¿Quién es inteligente? Encontrarlo utilizando resolución SLD y composición de sustituciones.

Utilizar los siguientes predicados y constantes: $R(X)$ para expresar que X es un robot, $Res(X,Y)$ para X puede resolver Y, $PL(X)$ para X es un problema lógico, $Pr(X)$ para X es un problema de esta práctica, $I(X)$ para X es inteligente, $J(X)$ para X es japonés y la constante alan para Alan.

Ejercicio 14 ★

Sean las siguientes cláusulas (en forma clausal), donde *suma* y *par* son predicados, *suc* es una función y *cero* una constante:

1. $\{\neg suma(X,Y,Z), suma(X, suc(Y), suc(Z))\}$ 2. $\{suma(X, cero, X)\}$ 3. $\{\neg suma(X,X,Y), par(Y)\}$

Demostrar utilizando resolución que suponiendo (1), (2), (3) se puede probar *par(suc(suc(cero)))*. Si es posible, aplicar resolución SLD. En caso contrario, utilizar resolución general. Mostrar en cada aplicación de la regla de resolución la sustitución utilizada.

Ejercicio 15

I. Pasar las siguientes fórmulas en lógica de primer orden a forma clausal.

a) $\forall C.(V(C) \vee \exists E.P(E,C))$ *b)* $\neg\exists C.(V(C) \wedge \exists E.P(E,C))$ *c)* $\forall E.\forall C.(P(E,i(C)) \Leftrightarrow P(E,C))$

II. A partir de las cláusulas definidas en el punto anterior, ¿puede demostrarse $\forall C.(V(i(C)) \Rightarrow V(C))$ usando resolución SLD? Si se puede, hacerlo. Si no, demostrarlo usando el método de resolución general.

Ejercicio 16 ★

Un lógico estaba sentado en un bar cuando se le ocurrió usar el método de resolución para demostrar el teorema del bebedor: siempre que haya alguien en el bar, habrá allí alguien tal que, si está bebiendo, todos en el bar están bebiendo. Sin embargo, el lógico en cuestión había bebido demasiado y la prueba no le salió muy bien. Esto fue lo que escribió en una servilleta del bar:

Teorema del bebedor:	$(\exists X.\texttt{enBar}(X)) \Rightarrow \exists Y.(\texttt{enBar}(Y) \wedge (\texttt{bebe}(Y) \Rightarrow \forall Z.(\texttt{enBar}(Z) \Rightarrow \texttt{bebe}(Z))))$
Elimino implicaciones:	$(\neg\exists X.\texttt{enBar}(X)) \vee \exists Y.(\texttt{enBar}(Y) \wedge (\neg\texttt{bebe}(Y) \vee \forall Z.(\neg\texttt{enBar}(Z) \vee \texttt{bebe}(Z))))$
Skolemizo:	$(\neg\texttt{enBar}(c)) \vee (\texttt{enBar}(k) \wedge (\neg\texttt{bebe}(k) \vee \forall Z.(\neg\texttt{enBar}(Z) \vee \texttt{bebe}(Z))))$
Paso a Forma Clausal:	1. $\{\neg\texttt{enBar}(c)\}$ 2. $\{\texttt{enBar}(k)\}$ 3. $\{\neg\texttt{bebe}(k)\}$ 4. $\{\neg\texttt{enBar}(Z), \texttt{bebe}(Z)\}$

Aplico resolución:

De 3 y 4 con $\sigma = \{k \leftarrow Z\}$:

5. $\{\neg\texttt{enBar}(Z)\}$

De 5 y 1 con $\sigma = \{Z \leftarrow c\}$:

□

a) Identificar los 5 errores cometidos en la demostración. (La fórmula original es correcta, notar que salteó pasos importantes e hizo mal otros).

b) Demostrar el teorema de manera correcta, usando resolución.

c) Indicar si la resolución utilizada en el punto b es o no SLD. Justificar.

Ejercicio 17

Dadas las siguientes afirmaciones:

- Toda persona tiene un contacto en Facebook:
  $\forall X.\exists Y.\mathsf{esContacto}(X,Y)$
  1. $\{\mathsf{esContacto}(X,\mathsf{f}(X))\}$
- La relación entre contactos es simétrica:
  $\forall X.\forall Y.(\mathsf{esContacto}(X,Y) \Rightarrow \mathsf{esContacto}(Y,X))$
  2. $\{\neg\mathsf{esContacto}(X,Y), \mathsf{esContacto}(Y,X)\}$

I. La siguiente es una demostración de que toda persona es contacto de sí misma, es decir, de que $\forall X\ \mathsf{esContacto}(X,X)$.

- Negando la conclusión:
  $\neg\forall X.\mathsf{esContacto}(X,X)$
- Forma normal negada:
  $\exists X.\neg\mathsf{esContacto}(X,X)$
- Skolemizando y en forma clausal:
  3. $\{\neg\mathsf{esContacto}(\mathsf{c},\mathsf{c})\}$
- De 1 y 3, con $\sigma = \{X := \mathsf{c}, \mathsf{f}(X) := \mathsf{c}\}$:
  □

¿Es correcta? Si no lo es, indicar el o los errores.

II. ¿Puede deducirse de las dos premisas que toda persona es contacto de alguien (es decir, de que $\forall Y.\exists X.\mathsf{esContacto}(X,Y)$)? En caso afirmativo dar una demostración, y en caso contrario explicar por qué.

Ejercicio 18 ★

Dadas las siguientes definiciones de Descendiente y Abuelo a partir de la relación Progenitor:

$\{\neg\text{Progenitor}(X,Y), \text{Descendiente}(Y,X)\}$ $\{\neg\text{Descendiente}(X,Y), \neg\text{Descendiente}(Y,Z), \text{Descendiente}(X,Z)\}$
$\{\neg\text{Abuelo}(X,Y), \text{Progenitor}(X, \text{medio}(X,Y))\}$ $\{\neg\text{Abuelo}(X,Y), \text{Progenitor}(\text{medio}(X,Y),Y)\}$

Demostrar usando resolución general que los nietos son descendientes; es decir, que

$$\forall X.\forall Y.(\text{Abuelo}(X,Y) \Rightarrow \text{Descendiente}(Y,X))$$

Ayuda: tratar de aplicar el método a ciegas puede traer problemas. Conviene tener en mente lo que se quiere demostrar.

Ejercicio 19 ★

En este ejercicio usaremos el método de resolución para demostrar una propiedad de las relaciones binarias; a saber, que una relación no vacía no puede ser a la vez irreflexiva, simétrica y transitiva.

Para esto se demostrará la propiedad deseada para una relación arbitraria R.

Dadas las siguientes definiciones:

1. R **es irreflexiva:** $\forall X.\neg R(X,X)$
2. R **es simétrica:** $\forall X.\forall Y.(R(X,Y) \Rightarrow R(Y,X))$
3. R **es transitiva:** $\forall X.\forall Y.\forall Z.((R(X,Y) \wedge R(Y,Z)) \Rightarrow R(X,Z))$
4. R **es vacía:** $\forall X.\neg\exists Y.R(X,Y)$

Utilizando resolución, demostrar que si R cumple las propiedades 1 a 3, entonces es vacía. Indicar si el método de resolución utilizado es o no SLD (Y justificar).

Ejercicio 20 ★

Considerar las siguientes definiciones en Prolog:

```
natural(cero).                  mayorOIgual(suc(X),Y) :- mayorOIgual(X, Y).
natural(suc(X)) :- natural(X).  mayorOIgual(X,X) :- natural(X).
```

- ¿Qué sucede al realizar la consulta `?- mayorOIgual(suc(suc(N)), suc(cero))`?
- Utilizar el método de resolución para probar la validez de la consulta del ítem 1. Para ello, convertir las cláusulas a forma clausal.
- Indicar si el método de resolución utilizado es o no SLD, y justificar. En caso de ser SLD, ¿respeta el orden en que Prolog hubiera resuelto la consulta?

Ejercicio 21

Dado el siguiente programa en Prolog, pasarlo a forma clausal y demostrar utilizando resolución que hay alguien que es inteligente pero analfabeto.

```
analfabeto(X) :- vivo(X), noSabeLeer(X).   noSabeLeer(X) :- mesa(X).
vivo(X) :- delfín(X).                      noSabeLeer(X) :- delfín(X).
inteligente(flipper).                      delfín(flipper).
inteligente(alan).
```

Ejercicio 22

Considerar las siguientes definiciones en prolog:

```
preorder(nil,[]).
preorder(bin(I,R,D),[R|L]) :- append(LI,LD,L),
                preorder(I,LI), preorder(D,LD).

append([],YS,YS).
append([X|XS],YS,[X|L]) :- append(XS,YS,L).
```

- ¿Qué sucede al realizar la consulta `?- preorder(bin(bin(nil,2,nil),1,nil),Lista).`?
- Utilizar el método de resolución para encontrar la solución al problema.
- Indicar si el método de resolución utilizado es o no SLD, y justificar. En caso de ser SLD, >respeta el orden en que Prolog hubiera resuelto la consulta?

Ejercicio 23

Dada la siguiente base de conocimientos en Prolog:

```
parPositivo(X,Y) :- mayor(X, 0), mayor(Y, 0).

natural(0).
natural(succ(N)) :- natural(N).

mayor(succ(X),0) :- natural(X).
mayor(succ(X),succ(Y)) :- mayor(X,Y).
```

a) Explicar con palabras qué sucede al realizar la siguiente consulta:

```
parPositivo(A,B), mayor(A,B).
```

b) Expresar la base de conocimientos y la consulta anterior como fórmulas lógicas, y luego encontrar una solución a la consulta utilizando resolución SLD.

Ejercicio 24

Sea la siguiente base de conocimientos en Prolog, que describe una parte de las reglas de reducción de un cierto lenguaje:

```
reduce(const * X * _, X).
reduce(id * X, X).
reduce(flip * F * X * Y, F * Y * X).
reduce(M * N, M1 * N) :- reduce(M, M1).
```

Donde el operador * representa la aplicación. Este operador asocia a izquierda. Si les resulta más cómodo, pueden reescribir las expresiones de la forma A * B como ap(A, B).

Se realiza la siguiente consulta:

```
? reduce(flip * const * X * Y, A), reduce(A, Z), reduce(const * id * X * Y, B), reduce(B,Z).
```

a) Reescribir la base de conocimientos y la consulta como fórmulas lógicas.

b) Resolver la consulta utilizando el método de resolución para obtener los valores de `A` y `B`.

c) La resolución utilizada en el punto anterior, ¿fue SLD? Justificar. En caso afirmativo, ¿fue la misma resolución que habría utilizado Prolog?

Ejercicio 25

El siguiente es un programa escrito en Prolog que define los números naturales, su relación de orden estricto, y un intento fallido de generar todos los pares de naturales.

```
natural(0).
natural(suc(X)) :- natural(X).

mayor(suc(X),X).
mayor(suc(X),Y) :- mayor(X,Y).

parDeNat(X,Y) :- natural(X), natural(Y).
```

Puede observarse que el programa no funciona correctamente, ya que, por ejemplo, la siguiente consulta se cuelga en lugar de arrojar una solución:

```
?- parDeNat(X,Y), mayor(X,Y).
```

Sin embargo, las definiciones son lógicamente correctas. Veámoslo usando resolución.

a) Convertir la base de conocimientos a forma clausal.

b) Utilizar el método de resolución para hallar una solución a la consulta.

c) La resolución realizada en el punto anterior ¿fue SLD? Justificar. En caso afirmativo, ¿en qué difiere de lo que habría hecho Prolog? En caso contrario, ¿sería posible encontrar una solución mediante resolución SLD? (No es necesario escribirla, solo justificar por qu'e es o no es posible.)

Ejercicio 26

a) Representar en forma clausal las siguientes fórmulas de lógica de primer orden referidas a números enteros.

   I. $\forall X.(par(X) \Rightarrow \exists Y.(Y > X \wedge \neg par(Y)))$ - Para todo X par existe un impar mayor que él.

   II. $\forall X.(\neg par(X) \Rightarrow \exists Y.(Y > X \wedge par(Y)))$ - Para todo X impar existe un par mayor que él.

   III. $\forall X.\forall Y.\forall Z.((X > Y \wedge Y > Z) \Rightarrow X > Z)$ - La relación de mayor es transitiva.

b) Usando resolución demostrar que para todo par existe otro par mayor, es decir, $\forall X.(par(X) \Rightarrow \exists Y.(Y > X \wedge par(Y)))$.

c) Indicar si la demostración es SLD y justificar.

Práctica Nº 9 - Programación Orientada a Objetos

Para resolver esta práctica, recomendamos usar el entorno *Pharo*, que puede bajarse del sitio web indicado en la sección *Enlaces* de la página de la materia.

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Objetos - Mensajes

Ejercicio 1 ★

En las siguientes expresiones, identificar mensajes. Indicar el objeto receptor y los colaboradores en cada caso.

a) `10 numberOfDigitsInBase: 2.`

b) `10 factorial.`

c) `20 + 3 * 5.`

d) `20 + (3 * 5).`

e) `December first, 1985.`

f) `1 = 2 ifTrue: [ 'what!?' ].`

g) `1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.`

h) `'Hello World' indexOf: $o startingAt: 6.`

i) `(OrderedCollection with: 1) add: 25; add: 35; yourself.`

j)
```
Object subclass: #SnakesAndLadders
instanceVariableNames: 'players squares turn die over'
classVariableNames: ''
poolDictionaries: ''
category: 'SnakesAndLadders'.
```

Ejercicio 2

Para cada una de las expresiones del punto anterior, indicar cuál es el resultado de su evaluación. Para este punto se recomienda utilizar el Workspace de *Pharo* para corraborar las respuestas.

Ejercicio 3

Dar ejemplos de expresiones válidas en el lenguaje *Smalltalk* que contengan los siguientes conceptos entre sus sub-expresiones. En cada caso indicar por qué se adapta a la categoría y describir que devuelve su evaluación.

a) Objeto

b) Mensaje unario

c) Mensaje binario

d) Mensaje keyword

e) Colaborador

f) Variable local

g) Asignación

h) Símbolo

i) Carácter

j) Array

Bloques - Métodos - Colecciones

Ejercicio 4 ★

Para cada una de las siguientes expresiones, indicar qué valor devuelve o explicar por qué se produce un error al ejecutarlas. Para este ejercicio recomendamos pensar qué resultado debería obtenerse y luego corraborarlo en el Workspace de Pharo.

a) `[:x | x + 1] value: 2`

b) `[|x| x := 10. x + 12] value`

c) `[:x :y | |z|  z := x + y] value: 1 value: 2`

d) `[:x :y | x + 1] value: 1`

e) `[:x | [:y | x + 1]] value: 2`

f) `[[:x | x + 1]] value`

g) `[:x :y :z | x + y + z] valueWithArguments: #(1 2 3)`

h) `[ |z| z := 10. [:x | x + z]] value value: 10`

¿Cuál es la diferencia entre `[|x y z| x + 1]` y `[:x :y :z| x + 1]`?

Ejercicio 5

Dada la siguiente implementación:

```
Integer << factorialsList
  |list|
  list :=  OrderedCollection with: 1.
  2 to: self do: [ :aNumber | list add:  (list last) * aNumber].
  ^list.
```

Donde `UnaClase << unMetodo` indica que se estará definiendo el método `#unMetodo` en la clase `UnaClase`.

¿Cuál es el resultado de evaluar las siguientes expresiones? ¿Quién es el receptor del mensaje `#factorialsList` en cada caso?

a) `factorialsList: 10.`

b) `Integer factorialsList: 10.`

c) `3 factorialsList.`

d) `5 factorialsList at: 4.`

e) `5 factorialsList at: 6.`

Ejercicio 6 ★

Mostrar un ejemplo por cada uno de los siguientes mensajes que pueden enviarse a las colecciones en el lenguaje Smalltalk. Indicar a qué evalúan dichos ejemplos.

a) `#collect:`

b) `#select:`

c) `#inject: into:`

d) `#reduce:` (o `#fold:`)

e) `#reduceRight:`

f) `#do:`

Ejercicio 7 ★

Suponiendo que tenemos un objeto *obj* que tiene el siguiente método definido en su clase

```
SomeClass << foo: x
  |aBlock z|
  z := 10.
  aBlock := [x > 5 ifTrue: [z := z + x. ^0] ifFalse: [z := z - x. 5]].
  y := aBlock value.
  y := y + z.
  ^y.
```

¿Cuál es el resultado de evaluar las siguientes expresiones?

a) `obj foo: 4.`

b) `Message selector: #foo: argument: 5.`

c) `obj foo: 10.` (Ayuda: el resultado no es 20).

Ejercicio 8 ★

Implementar métodos para los siguientes mensajes:

a) `#curry`, cuyo objeto receptor es un bloque de dos parámetros, y su resultado es un bloque similar al original pero "currificado".

Por ejemplo, la siguiente ejecución evalúa a 12.

```
|curried new|
curried := [ :x :res | x + res ] curry.
new := curried value: 10.
new value: 2.
```

b) `#flip`, que al enviarse a un bloque de dos parámetros, devuelve un bloque similar al original, pero con los parámetros en el orden inverso.

c) `#timesRepeat:`, cuyo objeto receptor es un número natural y recibe como colaborador un bloque, el cual se evaluará tantas veces como el número lo indique.

Por ejemplo, luego de la siguiente ejecución, `count` vale 20 y `copy` 18.

```
|count copy|
count := 0.
10 timesRepeat: [copy := count. count := count + 2].
```

Ejercicio 9 ★

Agregar a la clase `BlockClosure` el método de clase `generarBloqueInfinito` que devuelve un bloque `b1` tal que:

`b1 value` devuelve un arreglo de 2 elementos `#(1 b2)`
`b2 value` devuelve un arreglo de 2 elementos `#(2 b3)`
⋮
bi `value` devuelve un arreglo de 2 elementos `#(i` b_{i+1}`)`

Method Dispatch - Self - Super

Ejercicio 10

Indique en cada caso si la frase es cierta o falsa en *Smalltalk*. Si es falsa, ¿cómo podría corregirse?

I. Todo objeto es instancia de alguna clase y a su vez, estas son objetos.

II. Cuando un mensaje es enviado a un objeto, el método asociado en la clase del receptor es ejecutado.

III. Al mandar un mensaje a una clase, por ejemplo `Object new`, se busca en esa clase el método correspondiente. A este método lo clasificamos como método de instancia.

IV. Una *Variable de instancia* es una variable compartida por todas las instancias vivas de una clase, en caso de ser modificada por alguna de ellas, la variable cambia.

V. Las *Variables de clase* son accesibles por el objeto clase, pero al mismo tiempo también son accesibles y compartidas por todas las instancias de la clase; es decir, si una instacia modifica el valor de dicha variable, dicho cambio afecta a todas las instancias.

VI. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable *self*.

VII. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable *super*.

VIII. Un *Método de clase* puede acceder a las variables de clase pero no a las de instancia, y por otro lado, siempre devuelven un objeto instancia de la clase receptora.

IX. Los métodos y variables de clase son los métodos y variables de instancia del objeto clase.

Ejercicio 11 ★

Suponiendo que `anObject` es una instancia de la clase `OneClass` que tiene definido el método de instancia `aMessage`. Al ejecutar la siguiente expresión: `anObject aMessage`

I. ¿A qué objeto queda ligada (hace referencia) la pseudo-variable *self* en el contexto de ejecución del método que es invocado?

II. ¿A qué objeto queda ligada la pseudo-variable *super* en el contexto de ejecución del método que es invocado?

III. ¿Es cierto que `super == self`? ¿es cierto en cualquier contexto de ejecución?

Ejercicio 12

Se cuenta con la clase `Figura`, que tiene los siguientes métodos:

```
perimetro
    ^((self lados) sumarTodos).
lados
    ^self subclassResponsibility.
```

donde `sumarTodos` es un método de la clase `Collection`, que suma todos los elementos de la colección receptora. El método `lados` debe devolver un `Bag` (subclase de `Collection`) con las longitudes de los lados de la figura.

`Figura` tiene dos subclases: `Cuadrado` y `Círculo`. `Cuadrado` tiene una variable de instancia `lado`, que representa la longitud del lado del cuadrado modelado; `Círculo` tiene una variable de instancia `radio`, que representa el radio del círculo modelado.

Se pide que las clases `Cuadrado` y `Círculo` tengan definidos su método `perímetro`. Implementar los métodos que sean necesarios para ello, respetando el modelo (incompleto) recién presentado.

Observaciones: el perímetro de un círculo se obtiene calculando: $2 \cdot \pi \cdot radio$, y el del cuadrado: $4 \cdot lado$. Consideramos que un círculo no tiene lados. Aproximar π por 3,14.

Ejercicio 13 ★

```
Object subclass: Counter [
  |count| "Instance variable."
  class << new [
    ^super new initialize: 0.
  ]

  initialize: aValue [
    count := aValue.
    ^self.
  ]

  next [
    self initialize: count+1.
    ^count.
  ]

  nextIf: condition [
    ^condition ifTrue: [self next]
               ifFalse: [count]
    ]
]
```

```
Counter subclass: FlexibleCounter [
  |block| "Instance variable"
  class << new: aBlock [
    ^super new useBlock: aBlock.
  ]

  useBlock: aBlock [
    block := aBlock.
    ^self.
  ]

  next [
    self initialize: (block value: count).
    ^count.
  ]
]
```

En la siguiente expresión:

```
aCounter := FlexibleCounter new: [:v | v+2 ]. aCounter nextIf: true.
```

Se desea saber qué mensajes se envían a qué objetos (dentro del contexto de la clase) y cuál es el resultado de dicha evaluación. Recordar que := y ^ no son mensajes.

Recomendación, utilizar una tabla parecida a la siguiente:

Objeto	Mensaje	Resultado
FlexibleCounter	new:	un contador flexible (unCF de ahora en adelante)
...	...	...

Resolución para lógica de primer orden

Entrada: una fórmula σ de la lógica de primer orden.
Salida: un booleano indicando si σ es válida.
Si σ es válida, el método siempre termina.
Si σ no es válida, el método puede no terminar.

Método de resolución de primer orden (Procedimiento de semi-decisión)

1. Escribir $\neg\sigma$ como un conjunto $\mathcal{C}$ de **cláusulas**.
2. Buscar una **refutación** de $\mathcal{C}$.
   Si existe alguna refutación, el método encuentra alguna.
   Si no existe una refutación, el método puede "colgarse".

Pasaje a forma clausal en lógica de primer orden

Una fórmula se pasa a forma clausal aplicando las siguientes reglas.

Paso 1. Deshacerse del conectivo "$\Rightarrow$":

$$\sigma \Rightarrow \tau \quad \rightarrow \quad \neg\sigma \vee \tau$$

La fórmula resultante sólo usa los conectivos $\{\neg, \vee, \wedge, \forall, \exists\}$.

Paso 2. Empujar el conectivo "$\neg$" hacia adentro:

$$\begin{array}{lcl}
\neg(\sigma \wedge \tau) & \rightarrow & \neg\sigma \vee \neg\tau \\
\neg(\sigma \vee \tau) & \rightarrow & \neg\sigma \wedge \neg\tau \\
\neg\neg\sigma & \rightarrow & \sigma \\
\neg\forall \mathtt{X}.\, \sigma & \rightarrow & \exists \mathtt{X}.\, \neg\sigma \\
\neg\exists \mathtt{X}.\, \sigma & \rightarrow & \forall \mathtt{X}.\, \neg\sigma
\end{array}$$

La fórmula resultante está en **forma normal negada** (NNF):

$$\begin{array}{rcl}
\sigma_{\text{nnf}} & ::= & \mathbf{P}(t_1, \ldots t_n) \mid \neg\mathbf{P}(t_1, \ldots t_n) \mid \sigma_{\text{nnf}} \wedge \sigma_{\text{nnf}} \mid \sigma_{\text{nnf}} \vee \sigma_{\text{nnf}} \\
& \mid & \forall \mathtt{X}.\, \sigma_{\text{nnf}} \mid \exists \mathtt{X}.\, \sigma_{\text{nnf}}
\end{array}$$

Pasaje a forma clausal en lógica de primer orden

Paso 3. Extraer los cuantificadores ("$\forall/\exists$") hacia afuera.
Se asume siempre $\mathtt{X} \notin \mathrm{fv}(\tau)$:

$$
\begin{array}{llll}
(\forall \mathtt{X}.\, \sigma) \wedge \tau & \to \forall \mathtt{X}.\, (\sigma \wedge \tau) & \tau \wedge (\forall \mathtt{X}.\, \sigma) & \to \forall \mathtt{X}.\, (\tau \wedge \sigma) \\
(\forall \mathtt{X}.\, \sigma) \vee \tau & \to \forall \mathtt{X}.\, (\sigma \vee \tau) & \tau \vee (\forall \mathtt{X}.\, \sigma) & \to \forall \mathtt{X}.\, (\tau \vee \sigma) \\
(\exists \mathtt{X}.\, \sigma) \wedge \tau & \to \exists \mathtt{X}.\, (\sigma \wedge \tau) & \tau \wedge (\exists \mathtt{X}.\, \sigma) & \to \exists \mathtt{X}.\, (\tau \wedge \sigma) \\
(\exists \mathtt{X}.\, \sigma) \vee \tau & \to \exists \mathtt{X}.\, (\sigma \vee \tau) & \tau \vee (\exists \mathtt{X}.\, \sigma) & \to \exists \mathtt{X}.\, (\tau \vee \sigma)
\end{array}
$$

Todas las reglas transforman la fórmula en otra equivalente.

La fórmula resultante está en **forma normal prenexa**:

$$\sigma_{\mathrm{pre}} ::= \mathcal{Q}_1 \mathtt{X}_1.\, \mathcal{Q}_2 \mathtt{X}_2.\, \ldots\, \mathcal{Q}_n \mathtt{X}_n.\, \tau$$

donde cada $\mathcal{Q}_i$ es un cuantificador $\{\forall, \exists\}$
y τ representa una fórmula en NNF libre de cuantificadores.

Pasaje a forma clausal en lógica de primer orden

Paso 4. Deshacerse de los cuantificadores existenciales ($\exists$).
Para ello se usa la siguiente técnica de Herbrand y Skolem:

Lema (Skolemización)

$$\begin{array}{rcl}
\forall \mathtt{X}.\, \exists \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \text{ es sat.} & \text{sii} & \forall \mathtt{X}.\, \mathbf{P}(\mathtt{X}, \mathtt{f}(\mathtt{X})) \text{ es sat.} \\
\forall \mathtt{X}_1 \mathtt{X}_2.\, \exists \mathtt{Y}.\, \mathbf{P}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{Y}) \text{ es sat.} & \text{sii} & \forall \mathtt{X}_1 \mathtt{X}_2.\, \mathbf{P}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{f}(\mathtt{X}_1, \mathtt{X}_2)) \text{ es sat.} \\
 & \vdots & \\
\forall \vec{\mathtt{X}}.\, \exists \mathtt{Y}.\, \mathbf{P}(\vec{\mathtt{X}}, \mathtt{Y}) \text{ es sat.} & \text{sii} & \forall \vec{\mathtt{X}}.\, \mathbf{P}(\vec{\mathtt{X}}, \mathtt{f}(\vec{\mathtt{X}})) \text{ es sat.}
\end{array}$$

El lado izquierdo es una fórmula en el lenguaje $\mathcal{L}$.
El lado derecho es una fórmula el lenguaje $\mathcal{L} \cup \{\mathtt{f}\}$.

Caso particular cuando $|\vec{\mathtt{X}}| = 0$

$$\exists \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}) \text{ es sat.} \quad \text{sii} \quad \mathbf{P}(\mathtt{c}) \text{ es sat.}$$

El lenguaje se extiende con una nueva constante c.

Pasaje a forma clausal en lógica de primer orden

La Skolemización preserva la **satisfactibilidad**.
Pero no siempre produce fórmulas equivalentes.
Es decir **no preserva la validez**.

Ejemplo — la Skolemización no preserva la validez

$$\underbrace{\exists \mathtt{X}.\,(\mathbf{P}(0) \Rightarrow \mathbf{P}(\mathtt{X}))}_{\text{válida}} \qquad \underbrace{\mathbf{P}(0) \Rightarrow \mathbf{P}(\mathtt{c})}_{\text{inválida}}$$

Pasaje a forma clausal en lógica de primer orden

Dada una fórmula en forma normal prenexa, se aplica la regla:

$$\forall \mathtt{X}_1. \ldots \forall \mathtt{X}_n. \exists \mathtt{Y}. \sigma \quad \longrightarrow \quad \forall \mathtt{X}_1. \ldots \forall \mathtt{X}_n. \sigma\{\mathtt{Y} := \mathtt{f}(\mathtt{X}_1, \ldots, \mathtt{X}_n)\}$$

donde $\mathtt{f}$ es un símbolo de función nuevo de aridad $n \geq 0$.

La fórmula resultante está en **forma normal de Skolem**:

$$\sigma_{\mathrm{Sk}} ::= \forall \mathtt{X}_1 \mathtt{X}_2 \ldots \mathtt{X}_n. \tau$$

donde τ representa una fórmula en NNF libre de cuantificadores.

Pasaje a forma clausal en lógica de primer orden

Paso 5. Dada una fórmula en forma normal de Skolem:

$$\forall \mathtt{X}_1 \mathtt{X}_2 \ldots \mathtt{X}_n . \tau \quad (\tau \text{ libre de cuantificadores})$$

se pasa τ a forma normal conjuntiva usando las reglas ya vistas:

$$\begin{array}{lcl} \sigma \vee (\tau \wedge \rho) & \longrightarrow & (\sigma \vee \tau) \wedge (\sigma \vee \rho) \\ (\sigma \wedge \tau) \vee \rho & \longrightarrow & (\sigma \vee \rho) \wedge (\tau \vee \rho) \end{array}$$

El resultado es una fórmula de la forma:

$$\forall \mathtt{X}_1 \ldots \mathtt{X}_n . \left(\begin{array}{ll} & (\ell_1^{(1)} \vee \ldots \vee \ell_{m_1}^{(1)}) \\ \wedge & (\ell_1^{(2)} \vee \ldots \vee \ell_{m_2}^{(2)}) \\ \ldots & \\ \wedge & (\ell_1^{(k)} \vee \ldots \vee \ell_{m_k}^{(k)}) \end{array} \right)$$

Pasaje a forma clausal en lógica de primer orden

Paso 6. Empujar los cuantificadores universales hacia adentro:

$$\forall \mathtt{X}_1 \ldots \mathtt{X}_n. \left(\begin{array}{ll} & (\ell_1^{(1)} \vee \ldots \vee \ell_{m_1}^{(1)}) \\ \wedge & (\ell_1^{(2)} \vee \ldots \vee \ell_{m_2}^{(2)}) \\ & \ldots \\ \wedge & (\ell_1^{(k)} \vee \ldots \vee \ell_{m_k}^{(k)}) \end{array} \right) \longrightarrow \left(\begin{array}{ll} & \forall \mathtt{X}_1 \ldots \mathtt{X}_n.(\ell_1^{(1)} \vee \ldots \vee \ell_{m_1}^{(1)}) \\ \wedge & \forall \mathtt{X}_1 \ldots \mathtt{X}_n.(\ell_1^{(2)} \vee \ldots \vee \ell_{m_2}^{(2)}) \\ & \ldots \\ \wedge & \forall \mathtt{X}_1 \ldots \mathtt{X}_n.(\ell_1^{(k)} \vee \ldots \vee \ell_{m_k}^{(k)}) \end{array} \right)$$

Por último la **forma clausal** es:

$$\left\{ \begin{array}{l} \{\ell_1^{(1)}, \ldots, \ell_{m_1}^{(1)}\}, \\ \{\ell_1^{(2)}, \ldots, \ell_{m_2}^{(2)}\}, \\ \vdots \\ \{\ell_1^{(k)}, \ldots, \ell_{m_k}^{(k)}\} \end{array} \right\}$$

Pasaje a forma clausal en lógica de primer orden

Ejemplo — pasaje a forma clausal

Queremos ver que $\sigma \equiv \exists \mathtt{X}.\, (\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \Rightarrow \forall \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}, \mathtt{X}))$ es válida.
Primero la negamos: $\neg\sigma \equiv \neg\exists \mathtt{X}.\, (\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \Rightarrow \forall \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}, \mathtt{X}))$.
Pasamos $\neg\sigma$ a forma clausal:

$$
\begin{aligned}
& \neg\exists \mathtt{X}.\, (\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \Rightarrow \forall \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \neg\exists \mathtt{X}.\, (\neg\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \vee \forall \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, \neg(\neg\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \vee \forall \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, (\neg\neg\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \wedge \neg\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, (\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \wedge \exists \mathtt{Y}.\, \neg\mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, \exists \mathtt{Y}.\, (\forall \mathtt{Y}.\, \mathbf{P}(\mathtt{X}, \mathtt{Y}) \wedge \neg\mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, \exists \mathtt{Y}.\, \forall \mathtt{Z}.\, (\mathbf{P}(\mathtt{X}, \mathtt{Z}) \wedge \neg\mathbf{P}(\mathtt{Y}, \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, \forall \mathtt{Z}.\, (\mathbf{P}(\mathtt{X}, \mathtt{Z}) \wedge \neg\mathbf{P}(\mathtt{f}(\mathtt{X}), \mathtt{X})) \\
\rightarrow\ & \forall \mathtt{X}.\, \forall \mathtt{Z}.\, \mathbf{P}(\mathtt{X}, \mathtt{Z}) \wedge \forall \mathtt{X}.\, \forall \mathtt{Z}.\, \neg\mathbf{P}(\mathtt{f}(\mathtt{X}), \mathtt{X})
\end{aligned}
$$

La forma clausal es:

$$\{\{\mathbf{P}(\mathtt{X}, \mathtt{Z})\}, \{\neg\mathbf{P}(\mathtt{f}(\mathtt{X}), \mathtt{X})\}\} \equiv \{\{\mathbf{P}(\mathtt{X}, \mathtt{Y})\}, \{\neg\mathbf{P}(\mathtt{f}(\mathtt{Z}), \mathtt{Z})\}\}$$

PLP - Práctica 6: Resolución en Lógica

Gianfranco Zamboni

20 de marzo de 2018

Resolución en Lógica Proposicional

6.1. Ejercicio 1

Formula	FNC	FC
$p \supset p$	$\neg p \vee p$	$\{\neg p, p\}$
$(p \wedge q) \supset p$	$\neg p \vee \neg q \vee p$	$\{\neg p, \neg q, p\}$
$(p \vee q) \supset p$	$(\neg p \vee p) \wedge (\neg q \vee p)$	$\{\{\neg p, p\}, \{\neg q, p\}\}$
$\neg(p \iff \neg p)$	$\neg p \vee p$	$\{\{\neg p, p\}\}$
$\neg(p \wedge q) \supset (\neg p \vee \neg q)$	$(p \vee \neg p \vee \neg q) \wedge (q \vee \neg p \vee \neg q)$	$\{\{p, \neg p, \neg q\}, \{q, \neg p, \neg q\}\}$
$(p \wedge q) \vee (p \wedge r)$	$p \wedge (p \vee r) \wedge (q \vee p) \wedge (q \vee r)$	$\{\{p\}, \{p, r\}, \{q, p\}, \{q, r\}\}$
$(p \wedge q) \supset r$	$\neg p \vee \neg q \vee r$	$\{\neg p, \neg q, r\}$
$p \supset (q \supset r)$	$\neg p \vee \neg q \vee r$	$\{\neg p, \neg q, r\}$

6.2. Ejercicio 2

I. Para probar que una fórmula es una tatulogía hay que negarla, escribirla en forma clausal y ver que es insatisfactible agregando resolventes hasta llegar a la resolvente $\square$. Las tautologías son:

- $(p \supset p)$.

  Su negación $(p \wedge \neg p)$ escrita en forma clausal es $S = \{\{p\}, \{\neg p\}\}$. La resolvente entre $\{p\}$ y $\{\neg p\}\}$ es $\square$. Entonces S es insatisfacible.

- $(p \wedge q) \supset p$.

  Su negación $(p \wedge q \wedge \neg p)$ escrita en forma clausal es $S = \{\{p\}, \{q\}, \{\neg p\}\}$. Resolviendo $\{p\}$ y $\{\neg p\}\}$ obtenemos $\square$. Entonces S es insatisfactible.

- $\neg(p \iff \neg p)$.

  Su forma normal clausal es $(p \vee \neg p)$ y su negación $(p \wedge \neg p)$ que es el primer item, entonces es una tautología.

- $\neg(p \wedge q) \supset (\neg p \vee \neg q)$.

  Su negación $((\neg p \vee \neg q) \wedge p \wedge q$ escrita en forma clausal es $S = \{\{\neg p, \neg q\}, \{p\}, \{q\}\}$.

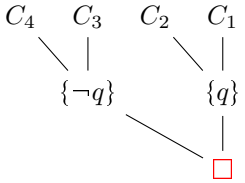
  De $\{p\}$ y $\{\neg p, \neg q\}$, obtenemos la resolvente $\{\neg q, \}$ y de $\{\neg q, \}$ y $\{\ q, \}$ obtenemos $\square$. Entonces es insatisfactible.

II. Queremos ver que $((\neg p \supset q) \wedge (p \supset q) \wedge (\neg p \supset \neg q)) \supset (p \wedge q)$. Osea que debemos negarla y usar resolución para ver que la fórmula negada es insatisfactible.

Paso a forma clausal de la fórmula:

$$\begin{aligned}
&\neg[((\neg p \supset q) \wedge (p \supset q) \wedge (\neg p \supset \neg q)) \supset (p \wedge q)] && \text{Negación}\\
&\neg[\neg((p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)) \vee (p \wedge q)] && \text{Eliminación de implicas}\\
&\neg\neg((p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)) \wedge \neg(p \wedge q) &&\\
&(p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \wedge \neg q) && \text{FNN y FNC}\\
&\{\underbrace{\{p,q\}}_{C_1}, \underbrace{\{\neg p,q\}}_{C_2}, \underbrace{\{p,\neg q\}}_{C_3}, \underbrace{\{\neg p,\neg q\}}_{C_4}\} && \text{Forma Clausal}
\end{aligned}$$

Resolución:



6.3. Ejercicio 3

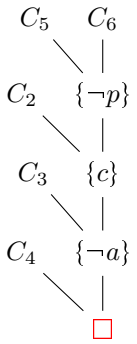
Valen las siguientes proposiciones:

- $p \supset a \rightsquigarrow \neg p \vee a \rightsquigarrow C_1 = \{\neg p, a\}$
- $\neg p \supset c \rightsquigarrow p \vee c \rightsquigarrow C_2 = \{p, c\}$
- $\neg(a \wedge c) \rightsquigarrow \neg a \vee \neg c \rightsquigarrow C_3 = \{\neg a, \neg c\}$
- $a \rightsquigarrow C_4 = \{a\}$
- $\neg l \rightsquigarrow C_5 = \{\neg l\}$

Queremos probar que vale $(p \wedge \neg l) \vee (\neg p \wedge l)$ usando resolución. Primero la negamos y la pasamos a forma clausal:

$$\begin{aligned}
&\neg[(p \wedge \neg l) \vee (\neg p \wedge l)] && \text{Negación}\\
&\neg(p \wedge \neg l) \wedge \neg(\neg p \wedge l) &&\\
&(\neg p \vee l) \wedge (p \vee \neg l) && \text{FNN y FNC}\\
&\{\underbrace{\{\neg p, l\}}_{C_6}, \underbrace{\{p, \neg l\}}_{C_7}\} && \text{Forma Clausal}
\end{aligned}$$

Y resolvemos:



Unificación en Lógica de Primer Orden

6.4. Ejercicio 4

1. $P(f(x))$ unifica con:
   a) $P(f(a))$ si $\sigma = \{a/x\}$
2. $P(a)$ unifica con:
   a) $P(x)$ si $\sigma = \{a/x\}$
3. $P(y)$ unifica con:
   a) $P(x)$ si $\sigma = \{y/x\}$
   b) $P(f(a))$ si $\sigma = \{f(a)/y\}$
4. $Q(x, f(y))$ unifica con:
   a) $Q(f(y), x)$ si $\sigma = \{f(y)/x\}$
5. $Q(x, f(z))$ unifica con:
   a) $Q(f(y), x)$ si $\sigma = \{f(y)/x,\ y/z\}$
   b) $Q(f(y), f(x))$ si $\sigma = \{f(y)/x,\ f(y)/z\}$
   c) $Q(f(y), y)$ si $\sigma = \{f(z)/y,\ f(f(z))/x\}$
6. $Q(x, f(a))$ unifica con:
   a) $Q(f(y), x)$ si $\sigma = \{a/y,\ f(a)/x\}$
   b) $Q(f(y), y)$ si $\sigma = \{f(a)/y,\ f(f(a))/x\}$

6.5. Ejercicio 5

1. $f(x, x, y) \doteq f(a, b, z) \underset{x/a}{\rightsquigarrow} f(a, a, y) \doteq f(a, b, z) \rightsquigarrow$ `falla` (a no unifica con b)
2. $f(x) \doteq y \underset{f(x)/y}{\rightsquigarrow} f(x) \doteq f(x)$ y el MGU es $\{f(x)/y\}$.
3. $f(g(c, y), x) \doteq f(z, g(z, a)) \underset{g(c,y)/z}{\rightsquigarrow} f(g(c, y), x) \doteq f(g(c, y), g(g(c, y), a))$

   $\underset{g(g(c,y),a)/x}{\rightsquigarrow} f(g(c, y), g(g(c, y), a)) \doteq f(g(c, y), g(g(c, y), a))$

   $$MGU = \{g(g(c, y), a)/x,\ g(c, y)/z\}$$
4. $f(a) \doteq g(y) \rightsquigarrow$ `falla` (f y g son funciones distintas).
5. $f(x) \doteq x \rightsquigarrow$ `falla` ($x \in FV(f(x))$)
6. $g(x, y) \doteq g(f(y), f(x)) \underset{f(y)/x}{\rightsquigarrow} g(f(y), y) \doteq g(f(y), f(f(y))) \rightsquigarrow$ `falla` ($y \in FV(f(f(y)))$)

Resolución en Lógica de Primer Orden

6.6. Ejercicio 6

I.

$$\forall x.\forall y.(\neg Q(x,y) \supset \neg P(x,y))$$
$$\forall x.\forall y.(\neg\neg Q(x,y) \vee \neg P(x,y))$$
$$\forall x.\forall y.(Q(x,y) \vee \neg P(x,y))$$

II.

$$\forall x.\forall y.((P(x,y) \wedge Q(x,y)) \supset R(x,y))$$
$$\forall x.\forall y.(\neg(P(x,y) \wedge Q(x,y)) \vee R(x,y))$$
$$\forall x.\forall y.(\neg P(x,y) \vee \neg Q(x,y) \vee R(x,y))$$

III.

$$\forall x.\exists y.(P(x,y) \supset Q(x,y))$$
$$\forall x.\exists y.(\neg P(x,y) \vee Q(x,y))$$

6.7. Ejercicio 7

I.

$$\exists x.\exists y.(x < y)$$
$$\exists y.(c < y)$$
$$c < d \quad \text{FSK}$$
$$\{\{c < d\}\}$$

II.

$$\forall x.\exists y.(x < y)$$
$$\forall x.(x < f(x)) \quad \text{FSK}$$
$$\{\{x < f(x)\}\} \quad \text{FC}$$

III.

$$\forall x.\neg(P(x) \wedge \forall y.(\neg P(y) \vee Q(y)))$$
$$\forall x.\neg P(x) \vee \neg\forall y.(\neg P(y) \vee Q(y))$$
$$\forall x.\neg P(x) \vee \exists y.\neg(\neg P(y) \vee Q(y))$$
$$\forall x.\neg P(x) \vee \exists y.(\neg\neg P(y) \wedge \neg Q(y))$$
$$\forall x.\neg P(x) \vee \exists y.(P(y) \wedge \neg Q(y))$$
$$\forall x.\neg P(x) \vee (P(c) \wedge \neg Q(c))$$
$$\forall x.(\neg P(x) \vee P(c)) \wedge (\neg P(x) \vee \neg Q(c))$$
$$\{\{\neg P(x), P(c)\}, \{\neg P(z), \neg Q(c)\}\}$$

IV.

$$\exists x.\forall y.(P(x,y) \wedge Q(x) \wedge \neg R(y)) \quad \text{FNN}$$
$$\forall y.(P(c,y) \wedge Q(c) \wedge \neg R(y))$$
$$\{\{P(c,y)\}, \{Q(c)\}, \{\neg R(z)\}\}$$

V.

$$\forall x.(P(x) \wedge \exists y.(Q(y) \vee \forall z.\exists w.(P(z) \wedge \neg Q(w))))$$
$$\forall x.(P(x) \wedge (Q(f(x)) \vee \forall z.\exists w.(P(z) \wedge \neg Q(w))))$$
$$\forall x.(P(x) \wedge (Q(f(x)) \vee \forall z.(P(z) \wedge \neg Q(f(x,z)))))$$
$$\forall x.\forall z.(P(x) \wedge (Q(f(x)) \vee (P(z) \wedge \neg Q(f(x,z)))))$$
$$\forall x.\forall z.(P(x) \wedge (Q(f(x)) \vee P(z)) \wedge (Q(f(x) \vee \neg Q(f(x,z))$$
$$\{\{P(x)\}, \{Q(f(x_1)), P(z)\}, \{Q(f(x_2)), \neg Q(f(x_2,z))\}\}$$

6.8. Ejercicio 8

I. La cláusula $\{p, \neg p\}$, si se resuelve consigo misma da la resolvente $\square$.

II. Las cláusulas $\{p, q\}$ y $\{\neg p, \neg q\}$ arrojan las siguientes resolventes:

- Si resolvemos los dos literales, entonces arroja $\square$.
- Si resolvemos solo con p, entonces arroja $\{q, \neg q\}$.
- Si resolvemos solo con q, entonces arroja $\{p, \neg p\}$.

III. Las clausulas de la forma$\{p, \neg p, q\}$ y $\{\neg p, p, \neg q\}$ deben unificar los tres literales al mismo tiempo para conseguir la clausula $\square$.

6.9. Ejercicio 9

I. $\exists x.\forall y.R(x,y) \supset \forall y.\exists x.R(x,y) \rightsquigarrow \exists x.\forall y.[R(x,y) \supset [\forall w.\exists z.R(z,w)]]$

Negación:

$$\begin{aligned}
&\neg\exists x.\forall y.[R(x,y) \supset [\forall w.\exists z.R(z,w)]]\\
&\neg\exists x.\forall y.[\neg R(x,y) \vee [\forall w.\exists z.R(z,w)]]\\
&\forall x.\neg\forall y.[\neg R(x,y) \vee [\forall w.\exists z.R(z,w)]]\\
&\forall x.\exists y.\neg[\neg R(x,y) \vee [\forall w.\exists z.R(z,w)]]\\
&\forall x.\exists y.[R(x,y) \wedge \neg[\forall w.\exists z.R(z,w)]]\\
&\forall x.\exists y.[R(x,y) \wedge [\exists w.\neg\exists z.R(z,w)]]\\
&\forall x.\exists y.[R(x,y) \wedge [\exists w.\forall z.\neg R(z,w)]]\\
&\forall x.\exists y.\exists w.\forall z.[R(x,y) \wedge \neg R(z,w)]\\
&\forall x.\exists w.\forall z.[R(x,f(x)) \wedge \neg R(z,w)]\\
&\forall x.\forall z.[R(x,f(x)) \wedge \neg R(z,g(x))]\\
&\{\{R(x_1,f(x_1))\},\{\neg R(z_2,g(x_2))\}\}
\end{aligned}$$

Resolución:

No podemos aplicar ningún paso de resolución a S porque las cláusulas no unifican: $R(x_1, f(x_1)) \doteq R(z, g(x_2)) \underset{x_1/x_2}{\rightsquigarrow}$ `falla` porque $f(x_2)$ y $g(x_2)$ no unifican.

II. $\forall x.\exists y.R(x,y) \supset \exists y.\forall x.R(x,y) \rightsquigarrow \forall x.\exists y.[R(x,y) \supset [\exists w.\forall z.R(z,w)]]$

Negación:

$$\begin{aligned}
&\neg\forall x.\exists y.[R(x,y) \supset [\exists w.\forall z.R(z,w)]]\\
&\neg\exists x.\exists y.[\neg R(x,y) \vee [\exists w.\forall z.R(z,w)]]\\
&\forall x.\neg\exists y.[\neg R(x,y) \vee [\exists w.\forall z.R(z,w)]]\\
&\exists x.\forall y.\neg[\neg R(x,y) \vee [\exists w.\forall z.R(z,w)]]\\
&\exists x.\forall y.[R(x,y) \wedge \neg[\exists w.\forall z.R(z,w)]]\\
&\exists x.\forall y.[R(x,y) \wedge [\forall w.\neg\forall z.R(z,w)]]\\
&\exists x.\forall y.[R(x,y) \wedge [\forall w.\exists z.\neg R(z,w)]]\\
&\exists x.\forall y.\forall w.\exists z.[R(x,y) \wedge \neg R(z,w)]\\
&\forall y.\forall w.\exists z.[R(c,y) \wedge \neg R(z,w)]\\
&\forall y.\forall w.[R(c,y) \wedge \neg R(f(y,w),w)]\\
&\{\{R(c,y_1)\},\{\neg R(f(y_2,w_2),w_2)\}\}
\end{aligned}$$

Resolución:

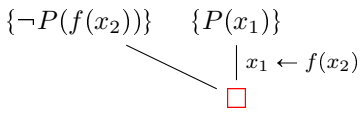
No podemos aplicar ningún paso de resolución a S porque c y $f(y_2, w_2)$ no unifican.

III. $\exists x.[P(x) \supset \forall x.P(x)] \rightsquigarrow \exists x.[P(x) \supset \forall y.P(y)]$

Negación:

$$\begin{aligned}
&\neg\exists x.[P(x) \supset \forall y.P(y)]\\
&\neg\exists x.[\neg P(x) \vee \forall y.P(y)]\\
&\forall x.\neg[\neg P(x) \vee \forall y.P(y)]\\
&\forall x.[P(x) \wedge \neg\forall y.P(y)]\\
&\forall x.[P(x) \wedge \neg\forall y.P(y)]\\
&\forall x.[P(x) \wedge \exists y.\neg P(y)]\\
&\forall x.[P(x) \wedge \exists y.\neg P(y)]\\
&\forall x.\exists y.[P(x) \wedge \neg P(y)]\\
&\forall x.[P(x) \wedge \neg P(f(x))]\\
&\{\{P(x_1)\}, \{\neg P(f(x_2))\}\}
\end{aligned}$$

Resolución:



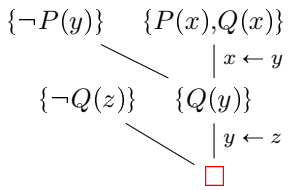
La negación de la formula es insatisfactible cuando $\sigma = \{f(x_2)/x\}$ por lo que la fórmula es valida.

IV. $\exists x.[P(x) \vee Q(x)] \supset [\exists x.P(x) \vee \exists x.Q(x)] \rightsquigarrow \exists x.[P(x) \vee Q(x)] \supset [\exists y.P(y) \vee \exists z.Q(z)]$

Negación:

$$\begin{aligned}
&\neg\exists x.[P(x) \vee Q(x)] \supset [\exists y.P(y) \vee \exists z.Q(z)]\\
&\neg\exists x.\neg[P(x) \vee Q(x)] \vee [\exists y.P(y) \vee \exists z.Q(z)]\\
&\forall x.\neg\neg[P(x) \vee Q(x)] \wedge \neg[\exists y.P(y) \vee \exists z.Q(z)]\\
&\forall x.[P(x) \vee Q(x)] \wedge [\forall y.\neg(P(y) \vee \exists z.Q(z))]\\
&\forall x.[P(x) \vee Q(x)] \wedge [\forall y.\neg P(y) \wedge \neg\exists z.Q(z)]\\
&\forall x.[P(x) \vee Q(x)] \wedge [\forall y.\neg P(y) \wedge \forall z.\neg Q(z)]\\
&\forall x.[P(x) \vee Q(x)] \wedge [\forall y.\neg P(y) \wedge \neg\exists z.Q(z)]\\
&\forall x.\forall y.\forall z.[P(x) \vee Q(x)] \wedge \neg P(y) \wedge \neg Q(z)\\
&\{\{P(x), Q(x)\}, \{\neg P(y)\}, \{\neg Q(z)\}\}
\end{aligned}$$

Resolución:



La negación de la formula es insatisfactible con $\sigma = \{z/y,\ z/x\}$ por lo que la fórmula es valida.

V. $\forall x.[P(x) \vee Q(x)] \supset [\forall x.P(x) \vee \forall x.Q(x)] \rightsquigarrow \forall x.[[P(x) \vee Q(x)] \supset [\forall y.[P(y) \vee \forall z.Q(z)]]]$

Negación:

$$\begin{aligned}
&\neg\forall x.[[P(x) \vee Q(x)] \supset [\forall y.[P(y) \vee \forall z.Q(z)]]]\\
&\neg\forall x.[\neg[P(x) \vee Q(x)] \vee [\forall y.[P(y) \vee \forall z.Q(z)]]]\\
&\exists x.[\neg\neg[P(x) \vee Q(x)] \wedge \neg[\forall y.[P(y) \vee \forall z.Q(z)]]]\\
&\exists x.[P(x) \vee Q(x)] \wedge [\exists y.\neg[P(y) \vee \forall z.Q(z)]]\\
&\exists x.[P(x) \vee Q(x)] \wedge [\exists y.\neg P(y) \wedge \neg\forall z.Q(z)]\\
&\exists x.[P(x) \vee Q(x)] \wedge [\exists y.\neg P(y) \wedge \exists z.\neg Q(z)]\\
&\exists x.\exists y.\exists z.[P(x) \vee Q(x)] \wedge [\neg P(y) \wedge \neg Q(z)]\\
&\exists y.\exists z.[P(c) \vee Q(c)] \wedge [\neg P(y) \wedge \neg Q(z)]\\
&\exists z.[P(c) \vee Q(c)] \wedge [\neg P(d) \wedge \neg Q(z)]\\
&[P(c) \vee Q(c)] \wedge \neg P(d) \wedge \neg Q(e)\\
&\{\{P(c), Q(c)\}, \{\neg P(d)\}, \{\neg Q(e)\}\}
\end{aligned}$$

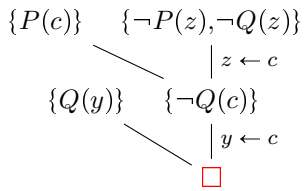
Resolución: Las clausulas no unifican entre si porque usan distintas constantes.

VI. $[\exists x.P(x) \land \forall x.Q(x)] \supset \exists x.[P(x) \land Q(x)] \rightsquigarrow [\exists x.P(x) \land \forall y.Q(y)] \supset \exists z.[P(z) \land Q(z)]$

Negación:

$$\begin{aligned}
&\neg([\exists x.P(x) \land \forall y.Q(y)] \supset \exists z.[P(z) \land Q(z)]) \\
&\neg(\neg[\exists x.P(x) \land \forall y.Q(y)] \lor \exists z.[P(z) \land Q(z)]) \\
&\neg\neg[\exists x.P(x) \land \forall y.Q(y)] \land \neg\exists z.[P(z) \land Q(z)]) \\
&[\exists x.P(x) \land \forall y.Q(y)] \land \forall z.\neg[P(z) \land Q(z)]) \\
&\exists x.P(x) \land \forall y.Q(y) \land \forall z.[\neg P(z) \lor \neg Q(z)]) \\
&\exists x.\forall y.\forall z.P(x) \land Q(y) \land [\neg P(z) \lor \neg Q(z)]) \\
&\forall y.\forall z.P(c) \land Q(y) \land [\neg P(z) \lor \neg Q(z)]) \\
&\{\{P(c)\}, \{Q(y)\}, \{\neg P(z), \neg Q(z)\}\}
\end{aligned}$$

Resolución:



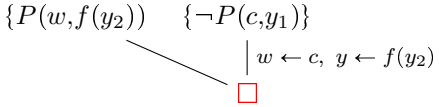
La negación de la formula es insatisfactible con $\sigma = \{c/y,\ c/z\}$ por lo que la fórmula es valida.

VII. $\forall x.\exists y.\forall z.\exists w.[P(x,y) \lor \neg P(w,z)]$

Negación:

$$\begin{aligned}
&\neg\forall x.\exists y.\forall z.\exists w.[P(x,y) \lor \neg P(w,z)] \\
&\exists x.\neg\exists y.\forall z.\exists w.[P(x,y) \lor \neg P(w,z)] \\
&\exists x.\forall y.\neg\forall z.\exists w.[P(x,y) \lor \neg P(w,z)] \\
&\exists x.\forall y.\exists z.\neg\exists w.[P(x,y) \lor \neg P(w,z)] \\
&\exists x.\forall y.\exists z.\forall w.\neg[P(x,y) \lor \neg P(w,z)] \\
&\exists x.\forall y.\exists z.\forall w.[\neg P(x,y) \land \neg\neg P(w,z)] \\
&\exists x.\forall y.\exists z.\forall w.[\neg P(x,y) \land P(w,z)] \\
&\forall y.\exists z.\forall w.[\neg P(c,y) \land P(w,z)] \\
&\forall y.\forall w.[\neg P(c,y) \land P(w,f(y))] \\
&\forall y.\forall w.[\neg P(c,y) \land P(w,f(y))] \\
&\{\{\neg P(c,y_1)\}, \{P(w,f(y_2))\}\}
\end{aligned}$$

Resolución:



La negación de la formula es insatisfactible con $\sigma = \{c/w,\ f(y_2)/y\}$ por lo que la fórmula es valida.

VIII. Este es muy largo, pero todos los literales terminan con constantes distintas y ninguno tiene variables.

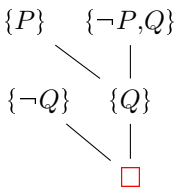
6.10. Ejercicio 10

I. Modus Ponens: $((P \supset Q) \land P) \supset Q$

Negación:

$$\begin{aligned}
&\neg[((P \supset Q) \land P) \supset Q] \\
&\neg[\neg((\neg P \lor Q) \land P) \lor Q] \\
&\neg\neg((\neg P \lor Q) \land P) \land \neg Q \\
&(\neg P \lor Q) \land P \land \neg Q \\
&\{\{\neg P, Q\}, \{P\}, \{Q\}\}
\end{aligned}$$

Resolución:

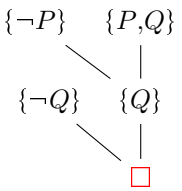


II. Modus Tollens: $((P \vee Q) \wedge \neg P) \supset Q$

Negación:

$$\begin{aligned} &\neg[((P \vee Q) \wedge \neg P) \supset Q] \\ &\neg[\neg((P \vee Q) \wedge \neg P) \vee Q] \\ &\neg\neg((P \vee Q) \wedge \neg P) \wedge \neg Q \\ &(P \vee Q) \wedge \neg P \wedge \neg Q \\ &\{\{P, Q\}, \{\neg P\}, \{\neg Q\}\} \end{aligned}$$

Resolución:



6.11. Ejercicio 11

I. $\{P(x), \neg P(x), Q(a)\}$ no es una cláusula de Horn.
Su fórmula de primer orden es $\forall x.P(x) \vee \neg P(x) \vee Q(a)$.

II. $\{P(x), \neg Q(y), \neg R(x, y)\}$ es una cláusula de de definición.
Su fórmula de primer orden es $\forall x \forall y.P(x) \vee \neg Q(y) \vee R(x, y)$

III. $\{\neg P(x, x, z), \neg Q(x, y), \neg Q(y, z)\}$ es una cláusula de Horn pero no de definición.
Su fórmula de primer orden: $\forall x.\forall z.\forall y.\neg P(x, x, z) \vee \neg Q(x, y) \vee \neg Q(y, z)$

IV. $\{M(1, 2, x)\}$ es una cláusula de definición.
Su fórmula de primer orden es $\forall x.M(1, 2, x)$.

6.12. Ejercicio 12

Condiciones son necesarias para que una demostración por resolución sea SLD:

1. Realizarse de manera lineal (utilizando en cada paso el resolvente obtenido en el paso anterior).
2. Utilizar únicamente cláusulas de Horn.
3. Empezar por una cláusula objetivo (sin literales positivos).
4. Empezar por una cláusula que provenga de la negación de lo que se quiere demostrar.
5. Utilizar la regla de resolución binaria en lugar de la general.

6.13. Ejercicio 13

Enunciado expresado en cláusulas:

- Alana es un robot japonés.

$$\begin{aligned} &R(\texttt{alan}) \wedge J(\texttt{alan}) \\ &\{\underbrace{\{R(\texttt{alan})\}}_{C_1}, \underbrace{\{J(\texttt{alan})\}}_{C_2}\} \end{aligned}$$

- Cualquier robot que puede resolver un problema lógico es `inteligente`.

$$\forall x.[R(x) \land \exists y.PL(y) \land Res(x,y)] \supset I(x)$$
$$\forall x.\neg[R(x) \land \exists y.PL(y) \land Res(x,y)] \lor I(x)$$
$$\forall x.\neg R(x) \lor \neg\exists y.PL(y) \lor \neg Res(x,y) \lor I(x)$$
$$\forall x.\neg R(x) \lor \neg\exists y.PL(y) \lor \neg Res(x,y) \lor I(x)$$
$$\forall x.\neg R(x) \lor \forall y.\neg PL(y) \lor \neg Res(x,y) \lor I(x)$$
$$\forall x.\forall y.\neg R(x) \lor \neg PL(y) \lor \neg Res(x,y) \lor I(x)$$
$$C_3 = \{\neg R(x_3), \neg PL(y_3), \neg Res(x_3,y_3), I(x_3)\}$$

- Todos los robots japoneses pueden resolver todos los problemas de esta práctica.

$$\forall x.[R(x) \land J(x)] \supset [\forall y.Pr(y) \land Res(x,y)]$$
$$\forall x.\neg[R(x) \land J(x)] \lor [\forall y.Pr(y) \land Res(x,y)]$$
$$\forall x.\forall y.[\neg R(x) \lor \neg J(x)] \lor [Pr(y) \land Res(x,y)]$$
$$\forall x.\forall y.[\neg R(x) \lor \neg J(x) \lor Pr(y)] \land [\neg R(x) \lor \neg J(x) \lor Res(x,y)]$$
$$\{\underbrace{\{\neg R(x_4), \neg J(x_4), Pr(y_4)\}}_{C_4}, \underbrace{\{\neg R(x_5), \neg J(x_5), Res(x_5,y_5)\}}_{C_5}\}$$

- Todos los problemas de esta práctica son lógicos.

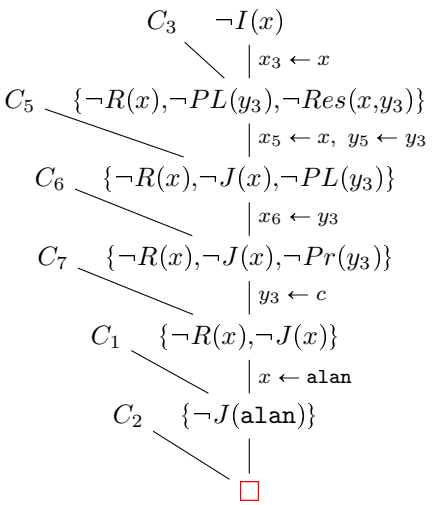
$$\forall x.Pr(x) \supset PL(x)$$
$$\forall x.\neg Pr(x) \lor PL(x)$$
$$C_6 = \{\neg Pr(x_6), PL(x_6)\}$$

- Existe al menos un problema en esta práctica.

$$\exists x.Pr(x)$$
$$Pr(c)$$
$$C_7 = \{Pr(c)\}$$

Queremos ver para que x vale $I(x)$, entonces lo negamos y usamos resolución para conseguir alguna sustitución que lo haga.

Resolución:



La sustitución resultante es $\sigma = \{alan/x,\ c/y3,\ c/x_6,\ alan/x_5,\ c/y_5,\ alan/x_3\}$. Entonces $alan$ es un robot `inteligente`.

6.14. Ejercicio 14

Cláusulas:

- $C_1 = \{\neg suma(x_1, y_1, z_1), suma(x_1, suc(y_1), suc(z_1))\}$
- $C_2 = \{suma(x_2, cero, x_2)\}$
- $C_3 = \{\neg suma(x_3, y_3, z_3), par(y_3)\}$

Resolución:

C_3 $\{\neg par(suc(suc(cero)))\}$
$\quad | \ y_3 \leftarrow suc(suc(cero))$
C_1 $\{\neg suma(x_3,suc(suc(cero)),z_3)\}$
$\quad | \ x_3 \leftarrow x_1,\ y_1 \leftarrow suc(cero), z_3 \leftarrow suc(z_1)$
C_1 $\{\neg suma(x_1,suc(cero),z_1)\}$
$\quad | \ x_1 \leftarrow x'_1,\ y'_1 \leftarrow cero, z_1 \leftarrow suc(z'_1)$
C_2 $\{\neg suma(x'_1,cero,z'_1)\}$
$\quad | \ x_2 \leftarrow x'_1,\ z'_1 \leftarrow x'_1$
$\square$

Y la sustitución resultante es

$$\sigma = \{x'_1/x_2,\ x'_1/z'_1,\ x'_1/x_1, cero/y'_1, suc(x'_1)/z_1, x'_1/x_3,\\ suc(cero)/y_1, suc(suc(x'_1))/z_3, suc(suc(cero))/y_3\}$$

La resolución usada es resolución SLD porque en cada paso resolvimos únicamente un literal (es lineal), comenzamos con un goal compuesto solo de negaciones y todas las cláusulas son fórmulas de Horn.

6.15. Ejercicio 15

I. Renombro las variables. $c \leftarrow x$ y $e \leftarrow y$ para que no sea confuso.

a) $\forall x.(V(x) \vee \exists y.(P(y, x)))$

$$\forall x.(V(x) \vee \exists y.(P(y, x)))$$
$$\forall x.\exists y.(V(x) \vee P(y, x))$$
$$C_1 = \{V(x_1), P(f(x_1), x_1)\}$$

b) $\neg\exists x.(V(x) \wedge \exists y.(P(y, x)))$

$$\neg\exists x.(V(x) \wedge \exists y.(P(y, x)))$$
$$\forall x.(\neg V(x) \vee \neg\exists y.(P(y, x)))$$
$$\forall x.(\neg V(x) \vee \forall y.\neg P(y, x))$$
$$\forall x.\forall y.(\neg V(x) \vee \neg P(y, x))$$
$$C_2 = \{\neg V(x_2), \neg P(y_2, x_2)\}$$

c) $\forall y.\forall x.(P(y, I(x)) \iff P(y, x))$

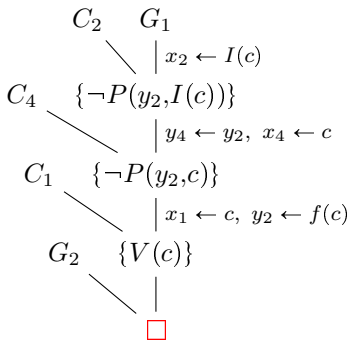
$$\begin{aligned}
&\forall y.\forall x.(P(y, I(x)) \iff P(y, x)) \\
&\forall y.\forall x.[P(y, I(x)) \supset P(y, x)] \wedge [P(y, x) \supset P(y, I(x))] \\
&\forall y.\forall x.[\neg P(y, I(x)) \vee P(y, x)] \wedge [\neg P(y, x) \vee P(y, I(x))] \\
&\{\underbrace{\{\neg P(y_3, I(x_3)), P(y_3, x_3)\}}_{C_3}, \underbrace{\{\neg P(y_4, x_4), P(y_4, I(x_4))\}}_{C_4}
\end{aligned}$$

II. Queremos ver que vale $\forall x.(V(I(x)) \supset V(x))$, entonces lo negamos, lo pasamos a forma clausal usamos resolución para inferir la insatisfactibilidad de la negación.

Negación:

$$\begin{aligned}
&\neg\forall x.(V(I(x)) \supset V(x)) \\
&\neg\forall x.(\neg V(I(x)) \vee V(x)) \\
&\exists x.(\neg\neg V(I(x)) \wedge \neg V(x)) \\
&\exists x.(V(I(x)) \wedge \neg V(x)) \\
&V(I(c)) \wedge \neg V(c) \\
&\{\underbrace{\{V(I(c))\}}_{G_1}, \underbrace{\{\neg V(c)\}}_{G_2}\}
\end{aligned}$$

No vamos a poder usar resolución SLD, por que la cláusula C_1 no es una fórmula de Horn (tiene más de un literal positivo) y hay cláusulas de nuestro Goal que no están negadas. **Resolución:**



Luego vale que $\forall x.(V(I(x)) \supset V(x))$.

6.16. Ejercicio 16

No lo pude demostrar.

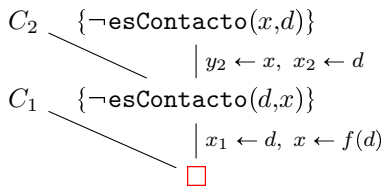
6.17. Ejercicio 17

- $C_1 = \{\texttt{esContacto}(x_1, f(x_1))\}$
- $C_2 = \{\neg\texttt{esContacto}(x_2, y_2), \texttt{esContacto}(y_2, x_2)\}$

I. La demostración no es correcta, en el último paso, realiza la sustitución $f(x) \leftarrow c$ pero $f(x)$ no unifica con c.

II. En el paso 4, la sustitución no es correcta, la resultante obtenida en ese paso debería ser $\{\neg\texttt{esContacto}(d, x)\}$.

III. Si, hay que seguir los mismos pasos que en el inciso anterior pero corrigiendo ese error:



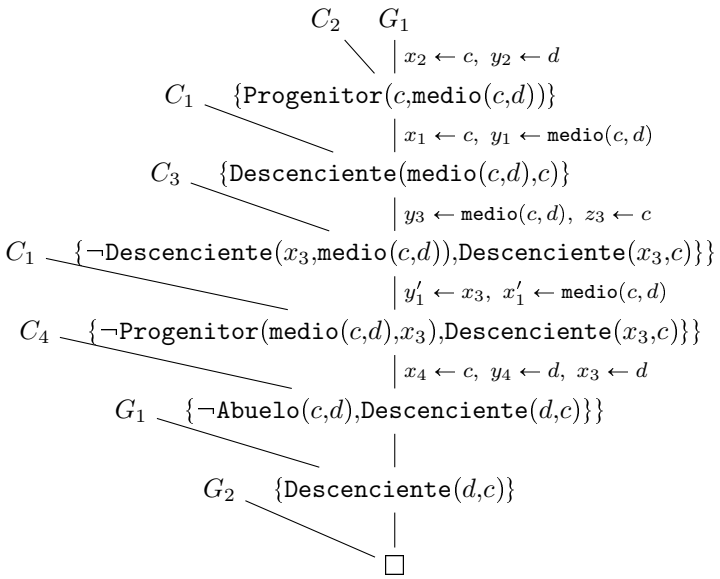
6.18. Ejercicio 18

- $C_1 = \{\neg\texttt{Progenitor}(x_1, y_1), \texttt{Descenciente}(y_1, x_1)\}$
- $C_2 = \{\neg\texttt{Abuelo}(x_2, y_2), \texttt{Progenitor}(x_2, \texttt{medio}(x_2, y_2))\}$
- $C_3 = \{\neg\texttt{Descenciente}(x_3, y_3), \neg\texttt{Descenciente}(y_3, z_3), \texttt{Descenciente}(x_3, z_3)\}$
- $C_4 = \{\neg\texttt{Abuelo}(x_4, y_4), \texttt{Progenitor}(\texttt{medio}(x_4, y_4), y_4)\}$

Queremos ver que: $\forall x.\forall y.(\texttt{Abuelo}(x, y) \supset \texttt{Descenciente}(y, x))$ **Negación:**

$$\neg\forall x.\forall y.(\texttt{Abuelo}(x, y) \supset \texttt{Descenciente}(y, x))$$
$$\neg\forall x.\forall y.(\neg\texttt{Abuelo}(x, y) \vee \texttt{Descenciente}(y, x)$$
$$\exists x.\neg\forall y.(\neg\texttt{Abuelo}(x, y) \vee \texttt{Descenciente}(y, x))$$
$$\exists x.\exists y.\neg(\neg\texttt{Abuelo}(x, y) \vee \texttt{Descenciente}(y, x))$$
$$\exists x.\exists y.(\neg\neg\texttt{Abuelo}(x, y) \wedge \neg\texttt{Descenciente}(y, x))$$
$$\exists x.\exists y.(\texttt{Abuelo}(x, y) \wedge \neg\texttt{Descenciente}(y, x))$$
$$\exists y.(\texttt{Abuelo}(c, y) \wedge \neg\texttt{Descenciente}(y, c))$$
$$\texttt{Abuelo}(c, d) \wedge \neg\texttt{Descenciente}(d, c)$$
$$\{\underbrace{\{\texttt{Abuelo}(c, d)\}}_{G_1}, \underbrace{\{\neg\texttt{Descenciente}(d, c)\}}_{G_2}\}$$

Resolución:



6.19. Ejercicio 19

- R es **irreflexiva**

$$\forall x.\neg R(x,x)$$
$$C_1 = \{\neg R(x_1, x_1)\}$$

- R es **símetrica**

$$\forall x.\forall y.(R(x,y) \supset R(y,x))$$
$$\forall x.\forall y.(\neg R(x,y) \vee R(y,x))$$
$$C_2 = \{\neg R(x_2, y_2), R(y_2, x_2)\}$$

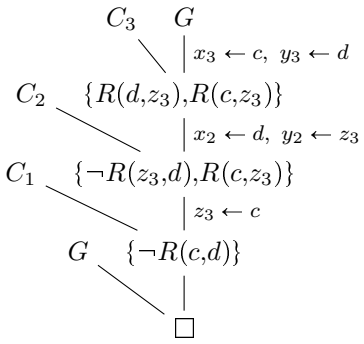
- R es **transitiva**

$$\forall x.\forall y.\forall z.((R(x,y) \wedge R(y,z)) \supset R(x,z))$$
$$\forall x.\forall y.\forall z.(\neg(R(x,y) \wedge R(y,z)) \vee R(x,z))$$
$$\forall x.\forall y.\forall z.((\neg R(x,y) \vee \neg R(y,z)) \vee R(x,z))$$
$$C_3 = \{\neg R(x_3, y_3), \neg R(y_3, z_3), R(x_3, z_3)\}$$

R es vacía es $\forall x.\neg\exists y.R(x,y)$, y queremos probrar que no existe una relación no vacia que pueda cumplir todas las propiedades a la vez, osea que $\neg(\forall x.\neg\exists y.R(x,y))$ es insatisfactible si valen las primeras tres propiedades. Entonces lo pasamos a forma clausal y usamos resolución para probarlo.

$$\neg(\forall x.\neg\exists y.R(x,y))$$
$$\exists x.\neg\forall y.\neg R(x,y)$$
$$\exists x.\exists y.\neg\neg R(x,y)$$
$$\exists x.\exists y.R(x,y)$$
$$\exists y.R(c,y)$$
$$R(c,d)$$
$$G = \{R(c,d)\}$$

Resolución:



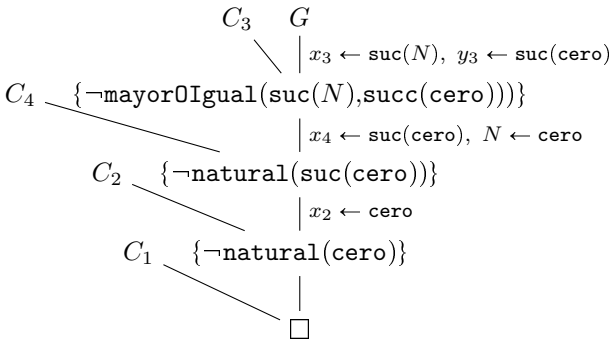
6.20. Ejercicio 20

I. El programa reducirá siempre usando la primer regla de `mayorOIgual`, sin embargo cuando llegue a la expresión `mayorOIgual(N,suc(cero))`, N no estará correctamente instanciada y podrá unificar con $succ(X)$ para cualquier X. Luego, intentará reducir la expresión `mayorOIgual(X,suc(cero))` y volverá a pasar lo mismo. Entonces el programa se cuelga.

II.

- $C_1 = \{\texttt{natural(cero)}\}$
- $C_2 = \{\texttt{natural(suc}(x_2)), \neg\texttt{natural}(x_2)\}$
- $C_3 = \{\texttt{mayorOIgual(suc}(x_3), y_3), \neg\texttt{mayorOIgual}(x_3, y_3)\}$
- $C_4 = \{\texttt{mayorOIgual}(x_4, x_4), \neg\texttt{natural}(x_4)\}$

El goal es $G = \{\neg\texttt{mayorOIgual(suc(suc}(N)), \texttt{suc(cero))}\}$



La sustitución resultado es: $\sigma = \{\texttt{cero}/x_2,\ \texttt{suc(cero)}/x_4,\ \texttt{cero}/N,\ \texttt{suc(cero)}/x_3,\ \texttt{suc(cero)}/y_3\}$ Por lo que $N = \texttt{cero}$ es una solución.

III. Es resolución SLD porque usamos cláusulas de horn, resolución binaria y lineal. Sin embargo, no usamos la misma técnica de selección que usa Prolog.

6.21. Ejercicio 21

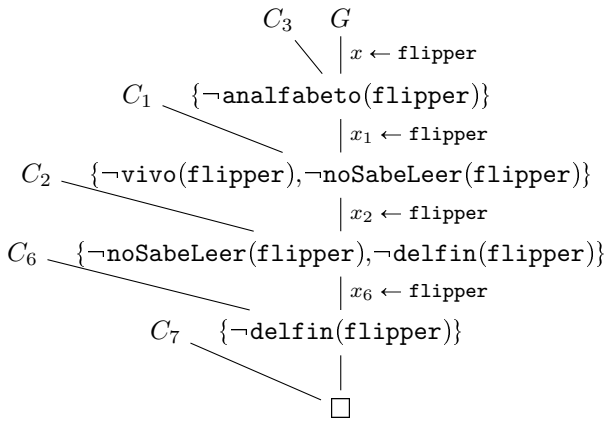
- $C_1 = \{\texttt{analfabeto}(x_1), \neg\texttt{vivo}(x_1), \neg\texttt{noSabeLeer}(x_1)\}$
- $C_2 = \{\texttt{vivo}(x_2), \neg\texttt{delfin}(x_2)\}$
- $C_3 = \{\texttt{inteligente(flipper)}\}$
- $C_4 = \{\texttt{inteligente(alan)}\}$
- $C_5 = \{\texttt{noSabeLeer}(x_5), \neg\texttt{mesa}(x_5)\}$
- $C_6 = \{\texttt{noSabeLeer}(x_6), \neg\texttt{delfin}(x_6)\}$
- $C_7 = \{\texttt{delfin(flipper)}\}$

Queremos probar que existe alguien inteligente pero analfabeto, es decir que $\exists x.\texttt{inteligente}(x) \wedge \texttt{analfabeto}(x)$.

Negación:

$$\neg\exists x.\texttt{inteligente}(x) \wedge \texttt{analfabeto}(x)$$
$$\forall x.\neg\texttt{inteligente}(x) \vee \neg\texttt{analfabeto}(x)$$
$$G = \{\neg\texttt{inteligente}(x) \vee \neg\texttt{analfabeto}(x)\}$$

Resolución:


C_3 G
$x \leftarrow$ flipper
C_1 {¬analfabeto(flipper)}
$x_1 \leftarrow$ flipper
C_2 {¬vivo(flipper),¬noSabeLeer(flipper)}
$x_2 \leftarrow$ flipper
C_6 {¬noSabeLeer(flipper),¬delfin(flipper)}
$x_6 \leftarrow$ flipper
C_7 {¬delfin(flipper)}
□

Último ejercicio

2° parcial 1° Cuat. 2011

En este ejercicio usaremos el método de resolución para demostrar una propiedad de las relaciones binarias; a saber, que una relación no vacía no puede ser a la vez irreflexiva, simétrica y transitiva.

Para esto tomaremos una relación R y se demostrará que, si R satisface las tres propiedades mencionadas, entonces es vacía.

Dadas las siguientes definiciones:

1. R **es irreflexiva:** $\forall X.\neg R(X, X)$
2. R **es simétrica:** $\forall X.\forall Y.(R(X, Y) \Rightarrow R(Y, X))$
3. R **es transitiva:** $\forall X.\forall Y.\forall Z.((R(X, Y) \wedge R(Y, Z)) \Rightarrow R(X, Z))$
4. R **es vacía:** $\forall X.\neg\exists Y.R(X, Y)$

Utilizando resolución, demostrar que sólo una relación vacía puede cumplir a la vez las propiedades 1 a 3. Indicar si el método de resolución utilizado es o no SLD (y justificar).

Último ejercicio

2° parcial 1° Cuat. 2011

Cast.:	R es irreflexiva.
1° o.:	$\forall X.\neg R(X, X)$
Claus.:	$\{\neg R(X_1, X_1)\}$
Cast.:	R es simétrica
1° o.:	$\forall X.\forall Y.(R(X, Y) \Rightarrow R(Y, X))$
Claus.:	$\{\neg R(X_2, Y_2), R(Y_2, X_2)\}$
Cast.:	R es transitiva.
1° o.:	$\forall X.\forall Y.\forall Z.((R(X, Y) \wedge R(Y, Z)) \Rightarrow R(X, Z))$
Claus.:	$\{\neg R(X_3, Y_3), \neg R(Y_3, Z_3), R(X_3, Z_3)\}$

Último ejercicio (cont.)

2° parcial 1° Cuat. 2011

Se desea demostrar que:

Cast.:	R es vacía:
1° o.:	$\forall X.\neg\exists Y.R(X,Y)$
Neg.:	$\exists X.\exists Y.R(X,Y)$
Claus.:	$\{R(\mathrm{a},\mathrm{b})\}$

Último ejercicio (resolviendo)

2° parcial 1° Cuat. 2011

1. $\{\neg R(X_1, X_1)\}$
2. $\{\neg R(X_2, Y_2), R(Y_2, X_2)\}$
3. $\{\neg R(X_3, Y_3), \neg R(Y_3, Z_3), R(X_3, Z_3)\}$
4. $\{R(\mathsf{a}, \mathsf{b})\}$
5. (4 y 2) $\{R(\mathsf{b}, \mathsf{a})\}$ $S = \{X_2 := \mathsf{a},\ Y_2 := \mathsf{b}\}$
6. (5 y 3) $\{\neg R(X_6, \mathsf{b}), R(X_6, \mathsf{a})\}$ $S = \{Y_3 := \mathsf{b},\ Z_3 := \mathsf{a}\}$ renombrando X_3 a X_6
7. (6 y 4) $\{R(\mathsf{a}, \mathsf{a})\}$ $S = \{X_6 := \mathsf{a}\}$
8. (7 y 1) $\square$ $S = \{X_1 := \mathsf{a}\}$

¿Esta demostración por resolución es SLD? ¿Por qué, o por qué no?

Alternativa SLD

2° parcial 1° Cuat. 2011

1. $\{\neg R(X_1, X_1)\}$
2. $\{\neg R(X_2, Y_2), R(Y_2, X_2)\}$
3. $\{\neg R(X_3, Y_3), \neg R(Y_3, Z_3), R(X_3, Z_3)\}$
4. $\{R(\mathsf{a}, \mathsf{b})\}$
5. (1 y 3) $\{\neg R(X_1, Y_3), \neg R(Y_3, X_1)\}$ $S = \{X_3 := X_1,\ Z_3 := X_1\}$
6. (5 y 4) $\{\neg R(\mathsf{b}, \mathsf{a})\}$ $S = \{X_1 := \mathsf{a},\ Y_3 := \mathsf{b}\}$
7. (6 y 2) $\{\neg R(\mathsf{a}, \mathsf{b})\}$ $S = \{X_2 := \mathsf{a},\ Y_2 := \mathsf{b}\}$
8. (7 y 4) $\square$ $S = \emptyset$

¿Es la única posible?

Otra alternativa SLD (más corta)

2° parcial 1° Cuat. 2011

1. $\{\neg R(X_1, X_1)\}$
2. $\{\neg R(X_2, Y_2), R(Y_2, X_2)\}$
3. $\{\neg R(X_3, Y_3), \neg R(Y_3, Z_3), R(X_3, Z_3)\}$
4. $\{R(\mathsf{a}, \mathsf{b})\}$
5. (1 y 3) $\{\neg R(X_1, Y_3), \neg R(Y_3, X_1)\}$ $S = \{X_3 := X_1,\ Z_3 := X_1\}$
6. (5 y 2) $\{\neg R(X_2, Y_2)\}$ $S = \{X_1 := X_2,\ Y_3 := Y_2\}$
7. (6 y 4) $\square$ $S = \{X_2 := \mathsf{a},\ Y_2 := \mathsf{b}\}$

Objetos

Considerar la siguiente clase que modela robots.

```
Object subclass: #Robot
  instanceVariableNames: 'x y b'
  ...

  Robot >> initWith: aBlock
    b := aBlock.
    x := 0.
    y := 0.
    ^ self.
```

```
Robot class >> newWith: aBlock
  |r|
  r := self new.
  ^ r initWith: aBlock.

Robot >> avanzar
  |res|
  res := b value: x value: y.
  x := res at: 1
  y := res at: 2
  ^ self.
```

Completar la definición de la clase `Drone` que es subclase de `Robot`.

Objetos

Los drones también se inicializan con un bloque que indica cómo debe avanzar sobre el plano x-y. Cada vez que avanza también se eleva una posición en el eje z (siempre y cuando no supere 10).

```
Robot subclass: #Drone
  instanceVariableNames: 'z'
  ...

  Drone class >> newWith: aBlock
    |r|
    r := super newWith: aBlock.
    ^ r init.
```

```
Drone >> init
  z := 0.
  ^ self.

Drone >> avanzar
  z < 10 ifTrue: [z := z+1].
  ^ super avanzar.
```

Indicar qué mensajes se envían a qué objetos, con qué colabradores y cuál es el resultado de cada colaboración al ejecutar la segunda línea del siguiente código:

```
aDrone := Drone newWith: [:n1 :n2 | #(n1+1 n2+1)].
aDrone avanzar.
```

- Definir el predicado `sublistaMasLargaDePrimos(+L,?P)` que es verdadero cuando `P` es una sublista de `L` que contiene la mayor cantidad de números primos consecutivos. Puede haber más de una solución. Por ejemplo:

```
?- sublistaMasLargaDePrimos([2,3,4,5,6,7,11],S).
S = [2,3];
S = [7,11];
false.
```

Programación Lógica

- Trabajaremos con árboles binarios, usando `nil` y `bin(I, N, D)` para representarlos en Prolog. Definir el predicado `listaDeÁrboles(-L)` que instancia en `L` **todas** las listas de árboles no vacíos (que no sean `nil`), con variables libres en sus nodos. Por ejemplo:

```
?- listaDeÁrboles(L).
L = [];
L = [bin(nil,_,nil)];
L = [bin(nil,_,nil), bin(nil,_,nil)];
L = [bin(nil,_, bin(nil,_,nil))];
L = [bin(bin(nil _,nil),_,nil)];
L = [bin(nil,_,nil), bin(nil,_,nil), bin(nil,_,nil)];
...
```

- El predicado del inciso anterior, ¿es reversible en `L`? Justificar.

Resolución Lógica - Inciso (a)

Convertir las siguientes fórmulas a Forma Clausal:

- $\forall C.(\texttt{camino}(C) \Leftrightarrow (\exists A.\exists B.\texttt{comunica}(A, B, C)))$
- $\forall X.\forall Y.\exists C.\texttt{comunica}(X, Y, C)$
- $\forall X.\forall Y.\forall C.(\texttt{comunica}(X, Y, C) \Rightarrow \texttt{conduceA}(Y, C))$
- $\forall X.\forall Y.\forall C.((\texttt{conduceA}(X, C) \wedge \exists D.\texttt{comunica}(X, Y, D)) \Rightarrow \texttt{conduceA}(Y, C))$

Resolución Lógica - Inciso (a)

$$\forall C.(camino(C) \iff (\exists A.\exists B.comunica(A,B,C))) \equiv$$
$$\forall C.[(camino(C) \Rightarrow \exists A_1.\exists B_1.comunica(A_1,B_1,C)) \wedge$$
$$((\exists A_2.\exists B_2.comunica(A_2,B_2,C)) \Rightarrow camino(C))] \equiv$$
$$\forall C.\exists A_1.\exists B_1.\forall A_2.\forall B_2.[(\neg camino(C) \vee comunica(A_1,B_1,C)) \wedge$$
$$(\neg comunica(A_2,B_2,C) \vee camino(C))] \rightarrow$$
$$\forall C.\forall A_2.\forall B_2.[(\neg camino(C) \vee comunica(f(C),g(C),C)) \wedge$$
$$(\neg comunica(A_2,B_2,C) \vee camino(C))] \equiv$$
$$\forall C.\forall A_2.\forall B_2.(\neg camino(C) \vee comunica(f(C),g(C),C)) \wedge$$
$$\forall C.\forall A_2.\forall B_2.(\neg comunica(A_2,B_2,C) \vee camino(C)) \equiv$$
$$\{\{\neg camino(C_1), comunica(f(C_1),g(C_1),C_1)\},$$
$$\{\neg comunica(A_2,B_2,C_2), camino(C_2)\}\}$$

Resolución Lógica - Inciso (b)

Utilizar el método de resolución para probar que todos los caminos conducen a Roma. Es decir:

$$\forall C.(\texttt{camino}(C) \Rightarrow \texttt{conduceA}(\mathit{Roma}, C))$$

(Notar que `conduceA` no significa "pasa por", sino "se puede extender para llegar a").

Resolución Lógica - Inciso (b)

① $\{\neg \texttt{camino}(C_1), \texttt{comunica}(f(C_1), g(C_1), C_1)\}$

② $\{\neg \texttt{comunica}(A_2, B_2, C_2), \texttt{camino}(C_2)\}$

③ $\{\texttt{comunica}(X_3, Y_3, h(X_3, Y_3))\}$

④ $\{\neg \texttt{comunica}(X_4, Y_4, C_4), \texttt{conduceA}(Y_4, C_4)\}$

⑤ $\{\neg \texttt{conduceA}(X_5, C_5), \neg \texttt{comunica}(X_5, Y_5, D_5), \texttt{conduceA}(Y_5, C_5)\}$

⑥ $\{\texttt{camino}(c)\}$

⑦ $\{\neg \texttt{conduceA}(Roma, c)\}$

Resolución Lógica - Inciso (b)

En lenguaje natural, las fórmulas que tenemos nos dicen:

- Si dos puntos X e Y están comunicados por C, C es un camino y visceversa.
- Para todo par de puntos X e Y, existe C que los comunica.
- Si un punto X y un punto Y son comunicados por C, en particular, C conduce a Y.
- Si C conduce a X, y existe D tal que comunica X e Y, entonces C conduce a Y.
- Existe un camino C que no conduce a Roma.

Resolución Lógica - Inciso (b)

Posible plan:
Como c es un camino que no conduce a Roma, el mismo no debe conducir a otro punto X tal que comunique con Roma por algún camino, ni conduce a este punto X. En particular c no comunica X con Roma, lo cual implica que c no es camino, absurdo! (Llegamos a la refutación).

① $\{\neg\texttt{camino}(C_1), \texttt{comunica}(f(C_1), g(C_1), C_1)\}$
② $\{\neg\texttt{comunica}(A_2, B_2, C_2), \texttt{camino}(C_2)\}$
③ $\{\texttt{comunica}(X_3, Y_3, h(X_3, Y_3))\}$
④ $\{\neg\texttt{comunica}(X_4, Y_4, C_4), \texttt{conduceA}(Y_4, C_4)\}$
⑤ $\{\neg\texttt{conduceA}(X_5, C_5), \neg\texttt{comunica}(X_5, Y_5, D_5), \texttt{conduceA}(Y_5, C_5)\}$
⑥ $\{\texttt{camino}(c)\}$
⑦ $\{\neg\texttt{conduceA}(Roma, c)\}$

Resolución Lógica - Inciso (b)

①$\{\neg\texttt{camino}(C_1), \texttt{comunica}(f(C_1), g(C_1), C_1)\}$
④$\{\neg\texttt{comunica}(X_4, Y_4, C_4), \texttt{conduceA}(Y_4, C_4)\}$
⑤$\{\neg\texttt{conduceA}(X_5, C_5), \neg\texttt{comunica}(X_5, Y_5, D_5), \texttt{conduceA}(Y_5, C_5)\}$
②$\{\neg\texttt{comunica}(A_2, B_2, C_2), \texttt{camino}(C_2)\}$ ⑥$\{\texttt{camino}(c)\}$
③$\{\texttt{comunica}(X_3, Y_3, h(X_3, Y_3))\}$ ⑦$\{\neg\texttt{conduceA}(Roma, c)\}$

- De 7 y 5 con $\sigma_1 = \{Y_5 \leftarrow Roma, C_5 \leftarrow c\}$, obtengo:
  ⑧ $\{\neg\texttt{conduceA}(X_5, c), \neg\texttt{comunica}(X_5, Roma, D_5)\}$
- De 8 y 4 con $\sigma_2 = \{Y_4 \leftarrow X_5, C_4 \leftarrow k\}$, obtengo:
  ⑨ $\{\neg\texttt{comunica}(X_4, X_5, c), \neg\texttt{comunica}(X_5, Roma, D_5)\}$
- De 9 y 1 con $\sigma_3 = \{X_4 \leftarrow f(c), X_5 \leftarrow g(c), C_1 \leftarrow c\}$, obtengo:
  ⑩ $\{\neg\texttt{camino}(c), \neg\texttt{comunica}(g(c), Roma, D_5)\}$
- 10 y 3 con $\sigma_4 = \{X_3 \leftarrow g(c), Y_3 \leftarrow Roma, D_5 \leftarrow h(g(c), Roma)\}$, obtengo: ⑪ $\{\neg\texttt{camino}(c)\}$
- De 11 y 6 con $\sigma_5 = \emptyset$, obtengo: ⑫ $\square$

Resolución Lógica - Inciso (c)

El método de resolución utilizado en el punto b), ¿fue SLD? Justificar.
Si, porque:

- Se utilizan solo cláusulas de Horn.
- Se empieza por una cláusula objetivo.
- Se realiza de manera lineal.
- Se utiliza la regla de resolución binaria en vez de la general.

Las tres leyes de la generación infinita

Para resolver un problema de generación infinita, es importante recordar estas tres leyes:

1. Nunca debe usarse más de un generador infinito.
2. El generador infinito siempre va a la izquierda de cualquier otro generador.
3. Los generadores infinitos deben usarse únicamente para generar infinitas soluciones[1].

Otra cosa a tener en cuenta: es importante es que las soluciones generadas en cada paso sean finitas, y que entre todas cubran todo el espacio de soluciones que se busca generar. Por ejemplo, si queremos generar todas las listas finitas de enteros positivos, no nos sirve que el generador infinito nos vaya dando la longitud de la lista, porque para cada longitud hay infinitas listas posibles. Tampoco nos sirve que vaya generando el primer elemento de la lista y que los demás estén acotados por el primero, porque entonces nunca generaríamos las listas cuyo primer elemento no es el máximo. En cambio la suma de los elementos de la lista sí es un buen valor para ir generando en cada paso, ya que hay una cantidad finita de listas para cada suma posible, y entre todas las sumas obtenemos todas las listas (además las soluciones en cada paso son disjuntas, lo cual siempre es útil, ya que no tenemos que ocuparnos de eliminar soluciones repetidas).

En caso de que haya más de una forma posible de ir recorriendo el espacio de búsqueda generando finitas soluciones en cada paso, conviene pensar cuál es la forma más sencilla. Por ejemplo, para generar árboles binarios, no es muy útil que el generator infinito genere la altura, ya que la altura de un subárbol no determina la del otro. No hay una receta para resolver todos los problemas posibles, pero si ven que el problema de cómo generar las soluciones en cada paso es muy complejo, probablemente haya otra forma de usar el generador infinito que divida el espacio en particiones más sencillas de generar.

[1] La tercera ley puede tener excepciones en Haskell si el problema se salva con la evaluación Lazy, pero siempre es importante asegurarse de que el programa no se cuelgue al terminar de generar las soluciones que queremos (y, obviamente, tampoco antes).

$\Gamma_1 = \forall x.\neg P(x), \exists x.P(x)$

$$\frac{\Gamma \vdash \exists X.\sigma \quad \Gamma, \sigma \vdash \tau \quad X \notin \mathsf{fv}(\Gamma, \tau)}{\Gamma \vdash \tau}\ \exists_e$$

tivos

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \exists x.P(x)} \quad \dfrac{}{\Gamma_1, P(x) \vdash P(x)}}{\forall x.\neg P(x), \exists x.P(x) \vdash P(x)}\ \exists_e \quad \dfrac{\dfrac{}{\forall x.\neg P(x), \exists x.P(x) \vdash \forall x.\neg P(x)}\ Ax}{\forall x.\neg P(x), \exists x.P(x) \vdash \neg P(x)}\ \forall_e}{\dfrac{\forall x.\neg P(x), \exists x.P(x) \vdash \bot}{\dfrac{\forall x.\neg P(x) \vdash \neg \exists x.P(x)}{\vdash \forall X.\neg P(X). \implies \neg \exists X.P(X)}\ \Rightarrow_i}\ \neg_i}\ \neg_e$$

τ $\{x := x\}$

$$\frac{\Gamma \vdash \forall X.\sigma}{\Gamma \vdash \sigma\{X := t\}}\ \forall_e$$

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash \neg\tau}{\Gamma \vdash \bot}\ \neg_e$$

$$\frac{\Gamma, \tau \vdash \bot}{\Gamma \vdash \neg\tau}\ \neg_i$$

```
#(1 2 3 4) collect: [ :each | each * 2 ]    ⟶    #( 2 4 6 8 )
#(1 2 3 4)
    inject: 0
    into: [ :each :result | each + result ]    ⟶    10

" testing "
#( 2 4 ) anySatisfy: [ :each | each odd ]    ⟶    false
#( 2 4 ) allSatisfy: [ :each | each even ]    ⟶    true

" finding "
'abcdef' includes: $e    ⟶    true
'abcdef' contains: [ :each | each isUppercase ]    ⟶    false
'abcdef'
    detect: [ :each | each isVowel ]
    ifNone: [ $u ]    ⟶    $a

" String – a collection of characters "
string := 'abc'.
string := string , 'DEF'    ⟶    'abcDEF'
string beginsWith: 'abc'    ⟶    true
string endsWith: 'abc'    ⟶    false
string includesSubString: 'cD'    ⟶    true
string asLowercase    ⟶    'abcdef
string asUppercase    ⟶    'ABCDEF'

" OrderedCollection – an ordered collection of objects "
ordered := OrderedCollection new.
ordered addLast: 'world'.
ordered addFirst: 'hello'.
ordered size    ⟶    2
ordered at: 2    ⟶    'world'
ordered removeLast    ⟶    'world'
ordered removeFirst    ⟶    'hello'
ordered isEmpty    ⟶    true

" Set – an unordered collection of objects without duplicates "
set := Set new.
set add 'hello'; add: 'hello'.
set size    ⟶    1

" Bag – an unordered collection of objects with duplicates "
bag := Bag new.
bag add: 'this'; add: 'that'; add: 'that'.
bag occurrencesOf: 'that'    ⟶    2
bag remove: 'that'.
bag occurrencesOf: 'that'    ⟶    1

" Dictionary – associates unique keys with objects "
dictionary := Dictionary new.
dictionary at: 'smalltalk' put: 80.
dictionary at: 'smalltalk'    ⟶    80
dictionary at: 'squeak' ifAbsent: [ 82 ]    ⟶    82
dictionary removeKey: 'smalltalk'.
dictionary isEmpty    ⟶    true
```

Streams

```
" ReadStream – to read a sequence of objects from a collection
    "
stream := 'Hello World' readStream.
stream next    ⟶    $H
stream upTo: $o    ⟶    'ell'
stream skip: 2.
stream peek    ⟶    $o
stream upToEnd    ⟶    'orld'

" WriteStream – to write a sequence of objects to a collection "
stream := WriteStream on: Array new.
stream nextPut: 'Hello'.
stream nextPutAll: #( 1 2 3 ).
stream contents    ⟶    #( 'Hello' 1 2 3 )
```

File Streams

```
fileStream := FileDirectory default newFileNamed: 'tmp.txt'.
fileStream nextPutAll: 'my cool stuff'.
fileStream close.

fileStream := FileDirectory default oldFileNamed: 'tmp.txt'.
fileStream contents    ⟶    'my cool stuff'
```

Method Definition

```
messageSelectorAndArgumentNames
    "comment stating purpose of message"
    | temporary variable names |
    statements
```

Class Definition

```
Object subclass: #NameOfSubclass
    instanceVariableNames: 'instVar1 instVar2'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Category-Name'
```

References

1. Andrew Black, Stéphane Ducasse, Oscar Nierstrasz and Damien Pollet, *Squeak by Example*, Square Bracket Associates, 2007, `squeakbyexample.org`.
2. Chris Rathman, *Terse guide to Squeak*, `wiki.squeak.org/squeak/5699`.
3. *Smalltalk*, Wikipedia, the free encyclopedia, `en.wikipedia.org/wiki/Smalltalk`.

Smalltalk Cheat Sheet

Software Composition Group
University of Bern

May 21, 2008

1. The Environment

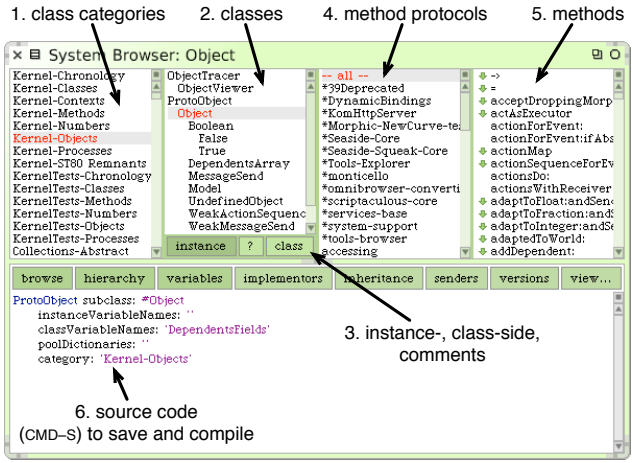


Figure 1: The Smalltalk Code Browser

- Do it (CMD–D): Evaluate selected code.
- Print it (CMD–P): Display the result of evaluating selected code.
- Debug it: Evaluate selected code step-by-step with the integrated debugger.
- Inspect it (CMD–I): Show an *object inspector* on the result of evaluating selected code.
- Explore it (CMD–SHIFT–I): Show an *object explorer* on the result of evaluating selected code.

2. The Language

- Everything is an object.
- Everything happens by sending messages.
- Single inheritance.
- Methods are public.
- Instance variables are private to objects.

Keywords

- self, the receiver.
- super, the receiver, method lookup starts in superclass.
- nil, the unique instance of the class UndefinedObject.
- true, the unique instance of the class True.
- false, the unique instance of the class False.
- thisContext, the current execution context.

Literals

- Integer
  123
  2r1111011 (123 in binary)
  16r7B (123 in hexadecimal)
- Float
  123.4
  1.23e−4
- Character
  $a
- String
  'abc'
- Symbol
  #abc
- Array
  #(123 123.4 $a 'abc' #abc)

Message Sends

1. *Unary messages* take no argument.
   1 factorial sends the message factorial to the object 1.
2. *Binary messages* take exactly one argument.
   3 + 4 sends message + with argument 4 to the object 3.
   #answer −> 42 sends −> with argument 42 to #answer.
   Binary selectors are built from one or more of the characters +, −, *, =, <, >, ...
3. *Keyword messages* take one or more arguments.
   2 raisedTo: 6 modulo: 10 sends the message named raisedTo:modulo: with arguments 6 and 10 to the object 2.

Unary messages are sent first, then binary messages and finally keyword messages:

2 raisedTo: 1 + 3 factorial ⟶ 128

Messages are sent left to right. Use parentheses to change the order:

1 + 2 * 3 ⟶ 9
1 + (2 * 3) ⟶ 7

Syntax

- Comments
  "Comments are enclosed in double quotes"
- Temporary Variables
  | var |
  | var1 var2 |
- Assignment
  var := aStatement
  var1 := var2 := aStatement
- Statements
  aStatement1. aStatement2
  aStatement1. aStatement2. aStatement3
- Messages
  receiver message (unary message)
  receiver + argument (binary message)
  receiver message: argument (keyword message)
  receiver message: argument1 with: argument2
- Cascade
  receiver message1; message2
  receiver message1; message2: arg2; message3: arg3
- Blocks
  [aStatement1. aStatement2]
  [:argument1 | aStatement1. aStatement2]
  [:argument1 :argument2 | | temp1 temp2 | aStatement1]
- Return Statement
  ^ aStatement

3. Standard Classes

Logical expressions

```
true not ⟶ false
1 = 2 or: [ 2 = 1 ] ⟶ false
1 < 2 and: [ 2 > 1 ] ⟶ true
```

Conditionals

```
1 = 2 ifTrue: [ Transcript show: '1 is equal to 2' ].
1 = 2 ifFalse: [ Transcript show: '1 isn''t equal to 2' ].

100 factorial / 99 factorial = 100
    ifTrue: [ Transcript show: 'condition evaluated to true' ]
    ifFalse: [ Beeper beep ].
```

Loops

```
" conditional iteration "
[ Sensor anyButtonPressed ]
    whileFalse: [ "wait" ].

pen := Pen newOnForm: Display.
pen place: Sensor cursorPoint.
[ Sensor anyButtonPressed ]
    whileTrue: [ pen goto: Sensor cursorPoint ].

" fixed iteration "
180 timesRepeat: [
    pen turn: 88.
    pen go: 250 ].

1 to: 100 do: [ :index |
    pen go: index * 4.
    bic turn: 89 ].

" infinite loop (press CMD+. to break) "
[ pen goto: Sensor cursorPoint ] repeat.
```

Blocks (anonymous functions)

```
" evaluation "
[ 1 + 2 ] value ⟶ 3
[ :x | x + 2 ] value: 1 ⟶ 3
[ :x :y | x + y ] value: 1 value: 2 ⟶ 3

" processes "
[ (Delay forDuration: 5 seconds) wait.
  Transcript show: 'done' ] fork ⟶ aProcess
```

Collections

```
" iterating "
'abc' do: [ :each | Transcript show: each ].
'abc'
    do: [ :each | Transcript show: each ]
    separatedBy: [ Transcript cr ].

" transforming "
#(1 2 3 4) select: [ :each | each even ] ⟶ #( 2 4 )
#(1 2 3 4) reject: [ :each | each = 2 ] ⟶ #( 1 3 4 )
```

Algoritmo de unificación

El algoritmo de unificación que conocíamos se adapta a términos de primer orden sólo cambiando la notación:

$$\{X \stackrel{?}{=} X\} \cup E \quad \xrightarrow{\texttt{Delete}} \quad E$$

$$\{\mathtt{f}(t_1, \ldots, t_n) \stackrel{?}{=} \mathtt{f}(s_1, \ldots, s_n)\} \cup E \quad \xrightarrow{\texttt{Decompose}} \quad \{t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n\} \cup E$$

$$\{t \stackrel{?}{=} X\} \cup E \quad \xrightarrow{\texttt{Swap}} \quad \{X \stackrel{?}{=} t\} \cup E \quad \text{si } t \text{ no es una variable}$$

$$\{X \stackrel{?}{=} t\} \cup E \quad \xrightarrow{\texttt{Elim}}_{\{X := t\}} \quad E\{X := t\} \quad \text{si } X \notin \mathrm{fv}(t)$$

$$\{\mathtt{f}(t_1, \ldots, t_n) \stackrel{?}{=} \mathtt{g}(s_1, \ldots, s_m)\} \cup E \quad \xrightarrow{\texttt{Clash}} \quad \texttt{falla} \quad \text{si } \mathtt{f} \neq \mathtt{g}$$

$$\{X \stackrel{?}{=} t\} \cup E \quad \xrightarrow{\texttt{Occurs-Check}} \quad \texttt{falla} \quad \text{si } X \neq t \text{ y } X \in \mathrm{fv}(t)$$

Justificacion de patrones de instanciacion y Reversibilidad en prolog

%prefijoHasta(?X, +L, ?Prefijo)
prefijoHasta(X, L, Prefijo) :- append(Prefijo, [X | _], L).

Si L no está instanciada, no hay manera de instanciarla, ya que el segundo argumento del append tiene un sufijo sin instanciar, y el tercero es la misma L que no está instanciada.
Si L está instanciada, tanto X como Prefijo pueden estar o no instanciados, porque append se encarga de instanciarlos si no lo están, o de verificar que coincidan con un prefijo de L y el elemento siguiente si ya están instanciados.

%desde(+X, -Y)
desde(X, X).
desde(X, Y) :- desde(X, Z), Y is Z + 1.

Si X no está instanciada, desde(X,Y) va a arrojar el resultado Y = X, y luego al entrar en la segunda cláusula va a arrojar un error al intentar realizar una operación aritmética sobre Z sin instanciar.
Si Y está instanciada, va a tener éxito si Y >= X, pero luego (o siempre, si Y < X) se va a colgar porque va a seguir generando infinitos valores para Z y comparando sus sucesores con Y, lo cual nunca va a tener éxito.

%desde2(+X,?Y)
desde2(X,X).
desde2(X,Y) :- nonvar(Y), X < Y.
desde2(X,Y) :- var(Y), desde2(X,Z), Y is Z + 1.

X debe estar instanciada por el mismo motivo que en desde/2.
Si Y no está instanciada, instancia Y en X para el primer resultado, y luego entra por la tercera cláusula y va generando infinitos valores para Y.
Si Y está instanciada, entra por la primera cláusula si es igual a X, y por la segunda en caso contrario, haciendo una comparación entre dos variables ya instanciadas, lo cual funciona correctamente.

%preorder(+A, ?L)
preorder(Nil, []).
preorder(Bin(I, R, D),[R | L]) :- preorder(I, LI), preorder(D, LD), append(LI, LD, L).

Si A no está instanciado, funciona para el caso L = [], porque solo unifica con la primera cláusula. Pero para L no vacía o no instanciada va a entrar (eventualmente) en la segunda cláusula, y va a llamar a preorder con dos variables sin instanciar, lo cual solo le va a permitir generar árboles Nil y listas vacías para luego volver a llamarse infinitamente con argumentos sin instanciar.
Si A está instanciado y L no, instancia L con [] si A es Nil, y en caso contrario calcula los respectivos preorders con I y D ya instanciados y los junta con append.
Si ambos argumentos están instanciados, utiliza unificación y/o append para verificar que L coincida con el preorder de A.
Predicados Prolog
Predicados: =, sort, msort, length, nth1, nth0, member, append, last, between, is_list, list_to_set, is_set, union, intersection, subset, subtract, select, delete, reverse, atom, number, numlist, sum_list, flatten
Operaciones extra-lógicas: is, \=, ==, =:=, =\=, >, <, =<, >=, abs, max, min, gcd, var, nonvar, ground, trace, notrace, make, halt

p(+A) indica que A debe proveerse instanciado.
p(-A) indica que A no debe estar instanciado.
p(?A) indica que A puede o no proveerse instanciado.

var(A) tiene ´exito si A es una variable libre.
nonvar(A) tiene ´exito si A no es una variable libre.
ground(A) tiene ´exito si A no contiene variables libres.

desde(X, X).
desde(X, Y) :- N is X+1, desde(N, Y).

Un predicado que usa el esquema G&T se define mediante otros dos:
pred(X1,...,Xn) :- generate(X1, ...,Xm), test(X1, ...,Xm).
Esta division de tareas implica que:
generate(...) deber´a instanciar ciertas variables.
test(...) deber´a verificar si los valores intanciados pertenecen a
la solucion, pudiendo para ello asumir que ya esta instanciada.

```
iesimo(0,[X|_],X).
iesimo(I,[_|XS],X) :- iesimo(I2,XS,X), I is I2 + 1.

desde2(X,X).
desde2(X,Y) :- var(Y), N is X+1, desde2(N,Y).
desde2(X,Y) :- nonvar(Y), X < Y.

pmq(X,Y) :- between(0,X,Y), Y mod 2 =:= 0.

paresSuman(S,X,Y) :- S1 is S-1, between(1,S1,X), Y is S-X.

generarPares(X,Y) :- desde2(2,S), paresSuman(S,X,Y).

coprimos(X,Y) :- generarPares(X,Y), gcd(X,Y) =:= 1.


altaMateria(plp).
altaMateria(aa).
altaMateria(metnum).

liviana(plp).
liviana(aa).
liviana(eci).

obligatoria(plp).
obligatoria(metnum).

leGusta(M) :- altaMateria(M).
leGusta(M) :- liviana(M).

hacer(M) :- leGusta(M), obligatoria(M).

hacerV2(M) :- setof(X,(leGusta(X),obligatoria(X)),L),
member(M,L).

% corteMasParejo(+L,-L1,-L2)
corteMasParejo(L,L1,L2) :- unCorte(L,L1,L2,D),
not((unCorte(L,_,_,D2), D2 < D)).
unCorte(L,L1,L2,D) :- append(L1,L2,L), sumlist(L1,S1),
sumlist(L2,S2), D is abs(S1-S2).
%
```

Implementar un predicado per´ımetro(?T,?P) que es verdadero cuando T es un tri´angulo y P es su per´ımetro. No se deben generar
resultados repetidos (no tendremos en cuenta la congruencia entre tri´angulos: si dos tri´angulos tienen las mismas longitudes, pero en diferente orden, se considerar´an diferentes entre s´ı). El predicado debe funcionar para cualquier instanciaci´on de T y P (no es necesario que funcione para tri´angulos parcialmente instanciados).
Implementar un generador de tri´angulos v´alidos, sin repetir resultados: tri´angulo(-T).
%

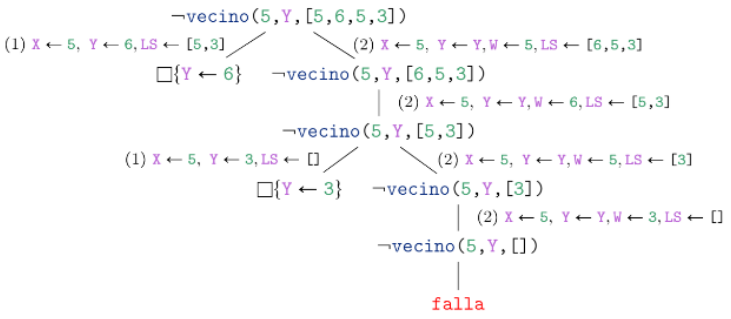
```
esTriangulo(tri(A,B,C)) :- A < B+C, B < A+C, C < B+A.


perimetro(tri(A,B,C),P) :- ground(tri(A,B,C)),
esTriangulo(tri(A,B,C)), P is A+B+C.
perimetro(tri(A,B,C),P) :- not(ground(tri(A,B,C))),
armarTriplas(P,A,B,C), esTriangulo(tri(A,B,C)).

armarTriplas(P,A,B,C) :- desde2(3,P), between(0,P,A), S is
P-A, between(0,S,B), C is S-B.

triangulos(T) :- perimetro(T,_).

%%Arbol de vecino:
vecino(X, Y, [X|[Y|Ls]]). vecino(X, Y, [W|Ls]) :- vecino(X, Y,
Ls).
vecino(5, Y, [5,6,5,3]):
```




```
%

desde(X,X).
desde(X,Y) :- N is X+1, desde(N,Y).


desde2(X,X).
desde2(X,Y) :- var(Y), N is X+1, desde2(N,Y).
desde2(X,Y) :- nonvar(Y), X < Y.
```

```
%TP
%% tablero(+Filas,+Columnas,-Tablero)
% lista(+Largo,-Lista)
lista(0,[]).
lista(N,[X|XS]):- length([X|XS], N), lista(Z ,XS) , X= _ , Z is N-1 .
tablero(0,_,[]).
tablero(F,C,[X|XS]):- length([X|XS],F) , tablero(F1 , C , XS) , X = X1 , lista(C,X1) , F1 is F-1 .


%% ocupar(+Pos,?Tablero)

% elementoTablero(+Pos,?Tablero,?Elemento)
elementoTablero(pos(0,0),[[X|_]|_],X).
elementoTablero(pos(0,C),[[_|XS]|YS],E):- 0 \= C , C1 is C-1 , elementoTablero(pos(0,C1),[XS|YS],E) .
elementoTablero(pos(F,C),[_|XS],E):- 0 \= F ,  F1 is F-1 , elementoTablero(pos(F1,C),XS,E).

ocupar(pos(F,C),T) :- elementoTablero(pos(F,C),T,ocupada).

%% Ejercicio 3
%% vecino(+Pos, +Tablero, -PosVecino)

vecino(pos(X,Y),[T|Ts],pos(F,Y)):- F is X+1, length([T|Ts],P), between(0,P,F).
vecino(pos(X,Y),[T|Ts],pos(F,Y)):- F is X-1, length([T|Ts],P), between(0,P,F).
vecino(pos(X,Y),[T|_],pos(X,J)):- J is Y+1, length(T,P), between(0,P,J).
vecino(pos(X,Y),[T|_],pos(X,J)):- J is Y-1, length(T,P), between(0,P,J).

 vecinoLibre(+Pos, +Tablero, -PosVecino)

vecinoLibre(pos(X,Y),[T|Ts],V) :- vecino(pos(X,Y),[T|Ts],V) , elementoTablero(V,[T|Ts],E),var(E).

%% camino(+Inicio, +Fin, +Tablero, -Camino) .
%caminoValido(+Inicio,+Fin,+Tablero,+Visitados,-Camino)
caminoValido(F,F,_,_,[]).
caminoValido(I,F,T,V,[P|C]):- not(I=F),vecinoLibre(I,T,P),not(member(P,V)),caminoValido(P,F,T,[P|V],C).
%camino(I,I,_,[]).
camino(I,F,T,[I|C]):- caminoValido(I,F,T,[I],C).

%% 5.1. Analizar la reversibilidad de los parámetros Fin y Camino justificando adecuadamente en cada
%% caso por qué el predicado se comporta como lo hace

% Analizando la reversibilidad de F, F al no venir instanciada unificara con la primera linea de camino valido,
% pero al solicitar otra solución en la siguiente linea ya no unificará,
% ya que "not(_ = F)" retornara siempre False. Por lo tanto, F no es reversible

%  En el caso de C, si esta instanciado, va a dar True si cada elemento efectivamente cumple con las condiciones que cada posicion del
%  tiene que cumplir en un camino valido, y que empieza y termine donde lo tiene que cumplir. Caso contrario da false
%  Por lo tanto C es reversible

%% camino2(+Inicio, +Fin, +Tablero, -Camino).

% desdeHasta(+Tablero , -Largo)
largoTablero([TH|TT],L) :- length([TH|TT],F) , length(TH,C) , L is F*C.

% desdeMenoresCaminos(+Tablero, -Camino)
desdeMenoresCaminos(T,C) :- largoTablero(T,F),
                            between(0,F,L),
                            length(C,L).

%camino2(I,I,_,[]).
camino2(I,F,T,[I|C]):- desdeMenoresCaminos(T,[I|C]), caminoValido(I,F,T,[I],C).

%% 6.1. Analizar la reversibilidad de los parámetros Inicio y Camino justificando adecuadamente en
%% cada caso por qué el predicado se comporta como lo hace.
% Cuando I si viene instanciada, ocurre un caso parecido al de F en camino, ya que unifica en el caso donde I es igual que F,
% pero luego al pedir la siguiente solucion da false por el "not(_ = I)". Entonces, I no es reversible.
% En C tambien es analogo al C de 5.1 , ya que devuelve True si cada elemento cumple las condiciones. En caso contrario da false.
% Por lo tanto C es reversible

%% caminoOptimo(+Inicio, +Fin, +Tablero, -Camino)
caminoOptimo(I,F,T,C):- camino(I,F,T,C), not((camino(I,F,T,C1) ,length(C,A),length(C1,B),B<A)).

%% caminoDual(+Inicio, +Fin, +Tablero1, +Tablero2, -Camino)

caminoDual(I, F , T1 , T2 , C):- camino(I,F,T1,C) , camino(I,F,T2,C).
```

```
sublistaMasLargaDePrimos(L,P) :-
sublistaDePrimosDeLong(L,P,Long),
not((sublistaDePrimosDeLong(L,_,Long2), Long2 > Long)).

sublistaDePrimosDeLong(L,P,Long) :- sublista(L,P),
soloPrimos(P), length(P,Long).

sublista(_,[]).
sublista(L,S) :- append(P,_,L), append(_,S,P), S \= [].

soloPrimos(L) :- not((member(X,L), not(esPrimo(X)))).

% esPrimo(+P)
esPrimo(P) :- P \= 1, P2 is P-1, not((between(2,P2,D),
mod(P,D) =:= 0)).


%?- listaDeÁrboles(L).
%L = [];

%L = [bin(nil,_,nil)]; ---> [1]
%L = [bin(nil,_,nil), bin(nil,_,nil)]; ---> [1,1]
%L = [bin(nil,_, bin(nil,_,nil))];  ---> [2]
%L = [bin(bin(nil _,nil),_,nil)];   ---> [2]
%L = [bin(nil,_,nil), bin(nil,_,nil), bin(nil,_,nil)]; --> [1,1,1]
%---> [1,2]
%---> [1,2]
%---> [2,1]
%---> [2,1]
%---> [3]

desde(X,X).
desde(X,Y) :- N is X+1, desde(N,Y).

listaDeArboles(L) :- desde(0,S), listaAcotadaDeArboles(S,L).

listaAcotadaDeArboles(0,[]).
listaAcotadaDeArboles(S,[X|XS]) :- between(1,S,Na),
                    arbolDeN(Na,X), S2 is S-Na,
                    listaAcotadaDeArboles(S2,XS).


arbolDeN(0,nil).
arbolDeN(N,bin(I,_,D)) :- N > 0, N2 is N-1,
paresQueSuman(N2,NI,ND), arbolDeN(NI,I),
arbolDeN(ND,D).

paresQueSuman(S,X,Y) :- between(0,S,X), Y is S-X.
```

```
%% Ejercicio 4

% juntar(?Lista1,?Lista2,?Lista3)
juntar([],Lista2,Lista2).
juntar([X|T1],Lista2,[X|T3]) :- juntar(T1,Lista2,T3).

%% Ejercicio 5

%% last(?L, ?U), donde U es el último elemento de la lista L
last([ X ], X).
last((_|T),Y) :- last(T,Y).

%mismaLongitud (+L,+L1)
mismaLongitud(L,L1) :- length(L,N) , length(L1,N).

%% reverse(+L, -L1), donde L1 contiene los mismos elementos que L, pero en orden inverso.
reverse([],[]).
reverse([X|XS],Y) :- mismaLongitud([X|XS],Y), reverse(XS,Z), append(Z, [X] , Y).


%% maxlista(+L , -M)  donde M es el maximo de cada lista.

maxlista([ X ] , X) .
maxlista([X,XS], X) :- maxlista(XS,Z), Z =< X.
maxlista([X,XS], Y) :- maxlista(XS,Y), X =< Y.

%% prefijo(?P , +L) donde S es el prefijo de la lista L
%prefijo([],[_]).
%prefijo([X],[X]).
prefijo([X],[X|_]).
prefijo([X|XS],[Y|YS]) :- prefijo(XS,YS) , X=Y.

%% sufijo(?S, +L), donde S es sufijo de la lista L.
%sufijo([],[_]).
%sufijo([X],[X]).
sufijo(X,Y) :- reverse(Y,Z), prefijo(Z2,Z) , reverse(Z2,X) .

%sublista(?S, +L), donde S es sublista de L.
sublista(X,Y) :- append(Z,_,Y) , append(_,X,Z) .

%pertenece(?X, +L), que es verdadero sii el elemento X se encuentra en la lista L. (Este predicado ya vienedefinido en Prolog y se llama member).
%pertenece(X,[X]).
pertenece(X,[X|_]).
pertenece(Y,[_|XS]) :- pertenece(Y,XS).


% Ejercicio 6

aplanar([],[]).
aplanar([ [] | T ], Res) :-
        aplanar(T, Res).
aplanar([ [X | T1 ] | T ], Res) :-
        aplanar([ X | T1 ], Y),
        aplanar(T, RecT),
        append(Y, RecT, Res).
aplanar([ X | T ], [X | Res]) :-
        not(is_list(X)),
        aplanar(T, Res).


% Ejercicio 7

%i. intersección(+L1, +L2, -L3)
%interseccionAux([L1H|L1T],L2,L3,V):- member(L1H,L2), member(L1H,L3) , not(member(L1H,V)) , interseccionAux (L1H,L2,L3,[L3H|V]).
%interseccion(L1,L2,L3):-interseccionAux(L1,L2,L3,[]).

% Ejercicio 12

% bin(izq, v, der),
vacio(nil).

raiz(bin(_, V, _),V).


altura(nil,0).
altura(bin(IZQ,_,DER),N) :- altura(IZQ,NI) , altura(DER,ND) , N is max(NI,ND) + 1 .

cantidadDeNodos(nil,0).
cantidadDeNodos(bin(IZQ,_,DER),N):- cantidadDeNodos(IZQ,NI) , cantidadDeNodos(DER,ND) , N is NI + ND + 1.

% ejemplo : funcion(bin(bin(bin(nil,2,nil),4,nil),5,bin(nil,6,nil)),X).

% Ejercicio 13

% inorder(+AB,-Lista)

inorder(nil,[]).
inorder(bin(IZQ,R,DER),L) :- inorder(IZQ,LI) , inorder(DER, LR) , append(LI,[R],L1) , append(L1,LR,L).

% no es mi sol
%arbolConInorder(-L, +AB)
arbolConInorder([], nil).
arbolConInorder(XS, bin(AI, X, AD)) :-
        reparto(XS, 2, [LI, [X | LD]]),
        arbolConInorder(LI, AI),
        arbolConInorder(LD, AD).
% ejemplo : arbolConInorder([2, 4, 5, 6],X).

% aBB(+T)


aBB(nil).
aBB(bin(nil,_,nil)).
aBB(bin(IZQ,R,nil)) :- raiz(IZQ,RI) , R >= RI , aBB(IZQ).
aBB(bin(nil,R,DER)):- raiz(DER,RD) , R =< RD , aBB(DER).
aBB(bin(IZQ,R,DER)):- raiz(IZQ,RI) , raiz(DER,RD) , R >= RI , R =< RD , aBB(IZQ) , aBB(DER).
```

```
% ejemplo aBB(bin(bin(bin(nil,2,bin(nil,3,nil)),4,nil),5,bin(nil,6,nil))).

% aBBInsertar(+X, +T1, -T2)
aBBInsertar(X,nil,bin(nil,X,nil)).
%aBBInsertar(X,bin(IZQ,X,DER),bin(IZQ,X,DER)). % por si son iguales , aunque no hace falta
aBBInsertar(X,bin(IZQ,R,DER),bin(IZQ2,R,DER)) :- X < R , aBBInsertar(X,IZQ,IZQ2).
aBBInsertar(X,bin(IZQ,R,DER),bin(IZQ,R,DER2)) :- X > R , aBBInsertar(X,DER,DER2).

% Ejercicio 15

desde2(X,X).
desde2(X,Y) :- var(Y), N is X+1, desde2(N,Y).
desde2(X,Y) :- nonvar(Y), X < Y.
todasLasFilasSumanLoMismo(XS) :- not((member(E1,XS), member(E2,XS) , E1 \= E2 , sumlist(E1,N1) , sumlist(E2,N2) , N1 \= N2 )).

cuadradoLat(N,M):- desde2(0,P), matrices(N,P,N,M).

matrices(_,_,0,[]).
matrices(N,P,C,[L|M]):- C > 0, generarLista(N,P,L) ,Y is C-1, matrices(N,P,Y,M).

generarLista(0,0 ,[]).
generarLista(N,P ,[X|XS]) :-
    N > 0, P >= 0,
    between(0, P, X),
    R is P - X, Y is N-1,
    generarLista(Y,R,XS).

%ahora con generate and test.
generarLista1(0,_ ,[]).
generarLista1(N,P ,[X|XS]) :-
    N > 0, P >= 0,
    between(0, P, X),
     Y is N-1,
    generarLista1(Y,P,XS).

cuadradoLat1(0,[]).
cuadradoLat1(N,M):- desde2(0,P), matrices1(N,P,N,M) , todasLasFilasSumanLoMismo(M).

matrices1(_,_,0,[]).
matrices1(N,P,C,[L|M]):- C > 0, generarLista1(N,P,L),Y is C-1, matrices1(N,P,Y,M).

% Parcial 1er cuat 2023

% palabra(+A,+N,-P)
palabra(_,0,[]).
palabra(A,N,[PH|PT]) :- N > 0 , I is N-1 , member(PH,A) , palabra(A,I,PT).

%frase(+A,-F).

generarFrases(_,0,[]).
generarFrases(A,P,[X|XS]):- between(1,P,Na), palabra(A,Na,X), S2 is P-Na, generarFrases(A,S2,XS).

frase(A,F) :- desde2(0,P), generarFrases(A,P,F).

% Estos ejs no son mios:

%borrar(+ListaOriginal, +X, -ListaSinXs)
borrar([], _, []).
borrar([X | T], X, ListaSinXs) :-
        borrar(T, X, ListaSinXs).
borrar([ Y | T], X, [ Y | Rec ]) :-
        X \= Y,
        borrar(T, X, Rec).
%sacarDuplicados(+L1, -L2)
sacarDuplicados([], []).
sacarDuplicados([ X | T], [ X | Rec ]) :-
        borrar(T, X, T1),
        sacarDuplicados(T1, Rec).
%concatenarTodas( +LL, -L)
concatenarTodas([], []).
concatenarTodas([ X | T], Res) :-
        concatenarTodas(T, Rec),
        concatenar(X, Rec, Res).
%todosSusMiembrosSonSublitas(+LListas, +L)
todosSusMiembrosSonSublitas([], _).
todosSusMiembrosSonSublitas([ X | XS], L) :-
        sublista(X, L),
        todosSusMiembrosSonSublitas(XS, L).
%reparto(+L, +N, -LListas)
reparto(L, N, LListas) :-
        length(LListas, N),
        todosSusMiembrosSonSublitas(LListas, L),
        concatenarTodas(LListas, L).
%repartoSinVacias(+L, -LListas)
repartoSinVacias(L, LListas) :-
        length(L, N),
        between(1, N, X),
        reparto(L, X, LListas),
        not(member([], LListas)).
```

-- Ejercicio 1

-- a. 10 numberOfDigitsInBase: 2

-- 10 es objeto receptor
-- numberOfDigitsInBase es mensaje
-- 2 es colaborador

-- b. 10 factorial

-- 10 es objeto receptor
-- factorial es mensaje

-- c. (20 + 3) * 5

-- 20 es objeto receptor
-- + es mensaje
-- 3 es colaborador
-- (20 + 3) es objeto receptor
-- * es mensaje
-- 5 es colaborador

-- d. 20 + (3 * 5)

-- 20 es objeto receptor
-- + es mensaje
-- el resultado de (3 * 5) es colaborador
-- 3 es objeto receptor
-- * es mensaje
-- 5 es colaborador

-- e. December first, 1985

-- Pharo no entiende el mensaje December, por lo que no se puede ejecutar

-- f. 1 = 2 ifTrue: ['what!?']

-- 1 es objeto receptor
-- = es mensaje
-- 2 es colaborador

-- el resultado de (1 = 2) es objeto receptor
-- ifTrue: es mensaje
-- ['what!?'] es colaborador

-- g. 1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.

-- 1@1 es objeto receptor
-- insideTriangle:with:with: es mensaje
-- 0@0, 2@0 y 0@2 son colaboradores

-- h. 'Hello World' indexOf: $o startingAt: 6

-- 'Hello World' es objeto receptor
-- indexOf:startingAt: es mensaje
-- $o y 6 son colaboradores

-- i. (OrderedCollection with: 1) add: 25; add: 35; yourself.

-- OrderedCollection with: 1 es objeto receptor
-- add: es mensaje
-- 25 es colaborador
-- add: es mensaje
-- 35 es colaborador
-- yourself es mensaje

-- j. Object subclass: #SnakesAndLadders instanceVariableNames: 'players squares turn die over' classVariableNames: '' poolDictionaries: '' category: 'SnakesAndLadders'

-- Object es objeto receptor (en este claso una clase)
-- subclass:instanceVariableNames:classVariableNames:poolDictionaries:category: es mensaje
-- #SnakesAndLadders, 'players squares turn die over', '', '', 'SnakesAndLadders' son colaboradores

-- Ejercicio 3

-- Mostrar expresiones válidas de Smallktalk que contengan los siguientes conceptos:

-- a) Objeto
-- 10

-- b) Mensaje unario
-- 10 factorial

-- c) Mensaje binario
-- 10 + 5

-- d) Mensaje keyword
-- 10 factorial printString

-- e) Colaborador
-- 10 + 5

-- f) Variable local
-- | x | x := 10

-- g) Asignación
-- x := 10

-- h) Símbolo
-- #hola

-- i) Carácter
-- $a

-- j) Array
-- #(1 2 3)

```
-- Ejercicio 4

-- Indicar el valor que devuelve cada una de las siguientes expresiones:

-- a) [:x | x + 1] value: 2
-- 3


-- b) [|x| x := 10. x + 12] value
-- 22


-- c) [:x :y | |z| z := x + y] value: 1 value: 2
-- 3


-- d) [:x :y | x + 1] value: 1
-- Exception: ArgumentsCountMismatch


-- e) [:x | [:y | x + 1]] value: 2
-- [:y | x + 1]


-- f) [[:x | x + 1]] value
-- [:x | x + 1]


-- g) [:x :y :z | x + y + z] valueWithArguments: #(1 2 3)
-- 6


-- h) [ |z| z := 10. [:x | x + z]] value value: 10
-- 20


-- ¿Cuál es la diferencia entre [|x y z| x + 1] y [:x :y :z| x + 1]?
-- En el primer caso hay tres variables locales, en el segundo caso hay tres argumentos


-- Ejercicio 5

-- Integer factorialsList
--   | list |
--   list := OrderedCollection with: 1.
--   2 to: self do: [:aNumber | list add: (list last) * aNumber ].
--   ^ list

-- a) factorialList: 10
-- el mensaje no tiene objeto receptor


-- b) Integer factorialsList: 10
-- la clase no entiende el mensaje


-- c) 3 factorialsList.
-- #(1 2 6 24)


-- d) 5 factorialsList at: 4
-- 24

-- e) 5 factorialsList at: 6
-- Exception: SubscriptOutOfBounds


-- Ejercicio 6

-- Mostrar un ejemplo por cada mensaje:

-- a) #collect:
-- #(1 2 3 4) collect: [:x | x + 1]
-- #(2 3 4 5)

-- b) #select:
-- #(1 2 3 4) select: [:x | x > 2]
-- #(3 4)

-- c) #inject: into: (how each element of the collection should be combined with the current accumulation to produce a new accumulation value)
-- "(#(1 2 3) inject: OrderedCollection new into: [ :a :e | a add: (e + 1). a ])
-- #(2 3 4)"

-- d) #reduce: (o #fold:)
-- #(1 2 3 4) reduce: [:x :y | x + y]
-- 10

-- e) #reduceRight: (it works like reduce but starts from the right like foldr in Haskell)
-- #(1 2 3 4) reduceRight: [:x :y | x - y]
-- -2

-- f) #do:
-- | accumulator | accumulator := 0. #(1 2 3 4) do: [:x | accumulator := accumulator + x]
-- 10


-- Ejercicio 7

-- SomeClass << foo: x
--  | aBlock z |
--  z := 10.
--  aBlock := [ x > 5 ifTrue: [ z := z + x. ^0 ] ifFalse: [ z := z = x. 5 ] ].
--  y := aBlock value.
--  y := y + z.
--  ^y.

-- a) obj foo: 4.
-- Instance of False did not understand #+

-- b) Message selector: #foo: argument: 5.
-- Es la instancia del mensaje foo con el argumento 5

-- c) obj foo: 10. (Ayuda: el resultado no es 20).
-- 0 (el resultado es 0 porque el bloque retorna 0 si x > 5 y no continúa con el resto de las instrucciones)


-- Ejercicio 8

-- a) #curry, cuyo objeto receptor es un bloque de dos parámetros,
-- y su resultado es un bloque similar al original pero currificado.

-- BlockClosure << curry
--   ^ [ :x | [ :y | self value: x value: y ] ] .


-- b) #flip, que al enviarse a un bloque de dos parámetros,
```

-- devuelve un bloque similar al original, pero con los parámetros en el orden inverso.

-- BlockClosure << flip
-- ^ [:x :y | self value: y value: x] .

-- c) #timesRepeat:, cuyo objeto receptor es un número natural y
-- recibe como colaborador un bloque, el cual se evaluará tantas veces como el número lo indique.

-- Integer << timesRepeat: aBlock
-- 1 to: self do: [aBlock value].

-- Ejercicio 9

-- Agregar a la clase BlockClosure el método de clase generarBloqueInfinito que devuelve un bloque b1 tal que:
-- b1 value devuelve un arreglo de 2 elementos #(1 b2),
-- b2 value devuelve un arreglo de 2 elementos #(2 b3),
-- ...,
-- bi value devuelve un arreglo de 2 elementos #(i bi+1)

-- BlockClosure class << generarBloqueInfinito
-- ^ [:x | #(x (self value: x + 1))] .

-- Ejercicio 10

-- i. Todo objeto es instancia de alguna clase y a su vez, estas son objetos.

-- [Verdadero]

-- ii. Cuando un mensaje es enviado a un objeto, el método asociado en la clase del receptor es ejecutado.

-- [Falso] El método asociado en la clase del receptor es ejecutado si el objeto entiende el mensaje, de lo
-- contrario se busca en la jerarquía de clases para encontrar la implementación del método correspondiente.

-- iii. Al mandar un mensaje a una clase, por ejemplo Object new, se busca en esa clase el método correspondiente.
-- A este método lo clasificamos como método de instancia.

-- [Falso] Al mandar un mensaje a una clase se busca el método de clase correspondiente.

-- iv. Una Variable de instancia es una variable compartida por todas las instancias vivas de una clase, en caso
-- de ser modificada por alguna de ellas, la variable cambia.

-- [Falso] Una Variable de instancia es una variable que pertenece a una instancia de una clase, por lo que no es
-- compartida por todas las instancias vivas de una clase.

-- v. Las Variables de clase son accesibles por el objeto clase, pero al mismo tiempo también son accesibles y
-- compartidas por todas las instancias de la clase; es decir, si una instacia modifica el valor de dicha variable,
-- dicho cambio afecta a todas las instancias.

-- [Falso] Las Variables de clase son accesibles por el objeto clase pero no significa que los valores sean compartidos.
-- Si una instancia modifica el valor de una variable de clase, el cambio no afecta a las demás instancias.

-- vi. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable self.

-- [Verdadero] Representará al objeto que recibe el mensaje.

-- vii. Al ver el código de un método, podemos determinar a qué objeto representará la pseudo-variable super.

-- [Verdadero] Representará a la superclase de la clase que implementa el método.

-- viii. Un Método de clase puede acceder a las variables de clase pero no a las de instancia, y por otro lado,
-- siempre devuelven un objeto instancia de la clase receptora.

-- [Falso] Un Método de clase puede acceder a las variables de clase y no a las de instancia, pero no siempre devuelven
-- un objeto instancia de la clase receptora.

-- ix. Los métodos y variables de clase son los métodos y variables de instancia del objeto clase.

-- [Falso] Los métodos y variables de clase son los métodos y variables de instancia del objeto clase.

-- Ejercicio 11

-- Suponiendo que anObject es una instancia de la clase OneClass que tiene definido el método de instancia
-- aMessage. Al ejecutar la siguiente expresión: anObject aMessage...

-- i. ¿A qué objeto queda ligada (hace referencia) la pseudo-variable self en el contexto de ejecución del método que es invocado?

-- El objeto anObject recibe el mensaje aMessage. En la definición del método de la instancia aMessage, self hace referencia a anObject.

-- ii. ¿A qué objeto queda ligada la pseudo-variable super en el contexto de ejecución del método que es invocado?

-- La pseudo-variable super quedará ligada a la superclase de OneClass.

-- iii. ¿Es cierto que super == self? ¿Es cierto en cualquier contexto de ejecución?

-- No es cierto que super == self. En cualquier contexto de ejecución super hace referencia a la superclase de la clase
-- que implementa el método, mientras que self hace referencia al objeto que recibe el mensaje.

-- Ejercicio 12

-- Se cuenta con la clase Figura, que tiene los métodos perimetro y lados.

-- sumarTodos es un método de la clase Collection, que suma todos los elementos de la colección receptora.
-- El método lados debe devolver un Bag (subclase de Collection) con las longitudes de los lados de la figura.

-- Figura tiene dos subclases: Cuadrado y Círculo. Cuadrado tiene una variable de instancia lado, que representa
-- la longitud del lado del cuadrado modelado; Círculo tiene una variable de instancia radio, que representa
-- el radio del círculo modelado.

-- Se pide que las clases Cuadrado y Círculo tengan definidos su método perímetro. Implementar los métodos
-- que sean necesarios para ello, respetando el modelo (incompleto) recién presentado.

-- Observaciones: el perímetro de un círculo se obtiene calculando: 2 · π · radio, y el del cuadrado: 4 · lado.
-- Consideramos que un círculo no tiene lados. Aproximar π por 3,14.

-- Object subclass: #Figura
-- instanceVariableNames: ''.

-- Figura >> perimetro
-- ^((self lados) sumarTodos).

-- Figura >> lados
-- self subclassResponsibility.

-- Figura subclass: #Cuadrado
-- instanceVariableNames: 'lado'.

-- Cuadrado >> lados
-- ^Bag with: (4 * lado).

-- Cuadrado >> perimetro
-- ^4 * lado.

-- Figura subclass: #Circulo
-- instanceVariableNames: 'radio'.

-- Circulo >> lados
-- ^Bag new. "Los círculos no tienen lados, devolvemos un Bag vacío"

-- Circulo >> perimetro
-- ^2 * 3.14 * radio.

-- Ejercicio 13

-- Object subclass: Counter [

-- | count |
-- class >> new [
-- ^ super new initialize: 0.
--]

-- initialize: aValue [
-- count := aValue.
-- ^ self.
--]

-- next [
-- self initialize: count + 1.
-- ^ count.
--]

-- nextIf: condition [
-- ^ condition ifTrue: [self next] ifFalse: [count]
--]
--]

-- Counter subclass: FlexibleCounter [

-- | block |
-- class >> new: aBlock [
-- ^ super new useBlock: aBlock.
--]

-- useBlock: aBlock [
-- block := aBlock.
-- ^ self.
--]

-- next [
-- self initialize: (block value: count).
-- ^ count.
--]
--]

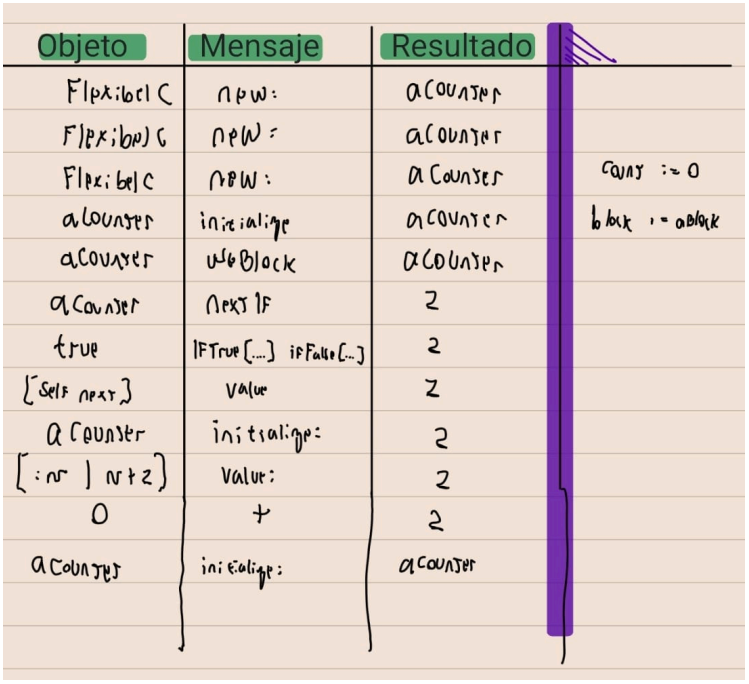
-- Considere la siguiente expresión: aCounter := FlexibleCounter new: [:v | v+2]. aCounter nextIf: true.

-- Se desea saber qué mensajes se envían a qué objetos (dentro del contexto de la clase) y cuál es el resultado
-- de dicha evaluación. Recordar que := y ^ no son mensajes. Recomendación, utilizar una tabla: "Objeto Mensaje Resultado"

-- Tenemos dos instrucciones, la primera crea una instancia de FlexibleCounter y la segunda envía el mensaje nextIf a la misma.

-- Objeto | Mensaje | Resultado
-- FlexibleCounter Class | new: | una instancia de FlexibleCounter
-- aCounter | nextIf: | 2

Mensajes comunes en colecciones:
add: agrega un elemento.
at: devuelve el elemento en una posici´on.
at:put: agrega un elemento a una posici´on.
includes: responde si un elemento pertenece o no.
includesKey: responde si una clave pertenece o no.
Paradigmas (de Lenguajes) de Programaci´on Clase pr´actica: Smalltalk 28 de junio de 2024 14 / 21
Colecciones
Mensajes m´as comunes
do: eval´ua un bloque con cada elemento de la colecci´on.
keysAndValuesDo: eval´ua un bloque con cada par clave-valor.
keysDo: eval´ua un bloque con cada clave.
select: devuelve los elementos de una colecci´on que cumplen un predicado (filter de funcional).
reject: la negaci´on del select:
collect: devuelve una colecci´on que es resultado de aplicarle un bloque a cada elemento de la colecci´on original (map de funcional).
detect: devuelve el primer elemento que cumple un predicado.
detect:ifNone: como detect:, pero permite ejecutar un bloque si no se encuentra ning´un elemento.
reduce: toma un bloque de dos o m´as par´ametros de entrada y hace
fold de los elementos de izquierda a derecha (foldl de funcional).



Metodos creados en Colecciones

```
minimo: aBlock
|b min |
min := 1000.
b:= self map1: aBlock.
b do: [ :each |  min > each ifTrue:[min := each ]].
^min.
map1: aBlock
|c|
c := OrderedCollection new.
self do: [ :each | c add: (aBlock value: each)].
^c
```

Minimo de Christian

```
minimo: aBlock
| minElement minValue |
self do: [:each | | val |
minValue ifNotNil: [
(val := aBlock value: each) < minValue ifTrue: [
minElement := each.
minValue := val]]
ifNil: ["first element"
minElement := each.
minValue := aBlock value: each].
].
^minElement
```

Dado el siguiente código:

```
drone := Drone newWith: [:n1 :n2 | {n1+1 . n2+1}].
drone avanzar.
```

se obtiene la tabla de seguimiento de abajo.

Objeto	Mensaje	Colaboradores	Ubicación del método	Resulta
Drone	newWith:	[:n1 :n2 ...]	Drone	aDrone
Drone	newWith:	[:n1 :n2 ...]	Robot	aDrone
Drone	new	-	Object	aDrone
aDrone	initWith:	[:n1 :n2 ...]	Robot	aDrone
aDrone	init	-	Drone	aDrone
aDrone	avanzar	-	Drone	aDrone
0	<	10	SmallInteger	True
true	ifTrue:	[z:=(z+1)]	True	1
[z:=(z+1)]	value	-	BlockClosure	1
0	+	1	SmallInteger	1
aDrone	avanzar	-	Robot	aDrone
[:n1 :n2 ...]	value: value:	0, 0	BlockClosure	#(1 1)
0	+	1	SmallInteger	1
0	+	1	SmallInteger	1
#(1 1)	at:	1	OrderedCollection	1
#(1 1)	at:	2	OrderedCollection	1