Razonamiento en sistemas de conocimiento basados en lógica

José A. Alonso y Francisco J. Martín

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Programa lógico y pregunta

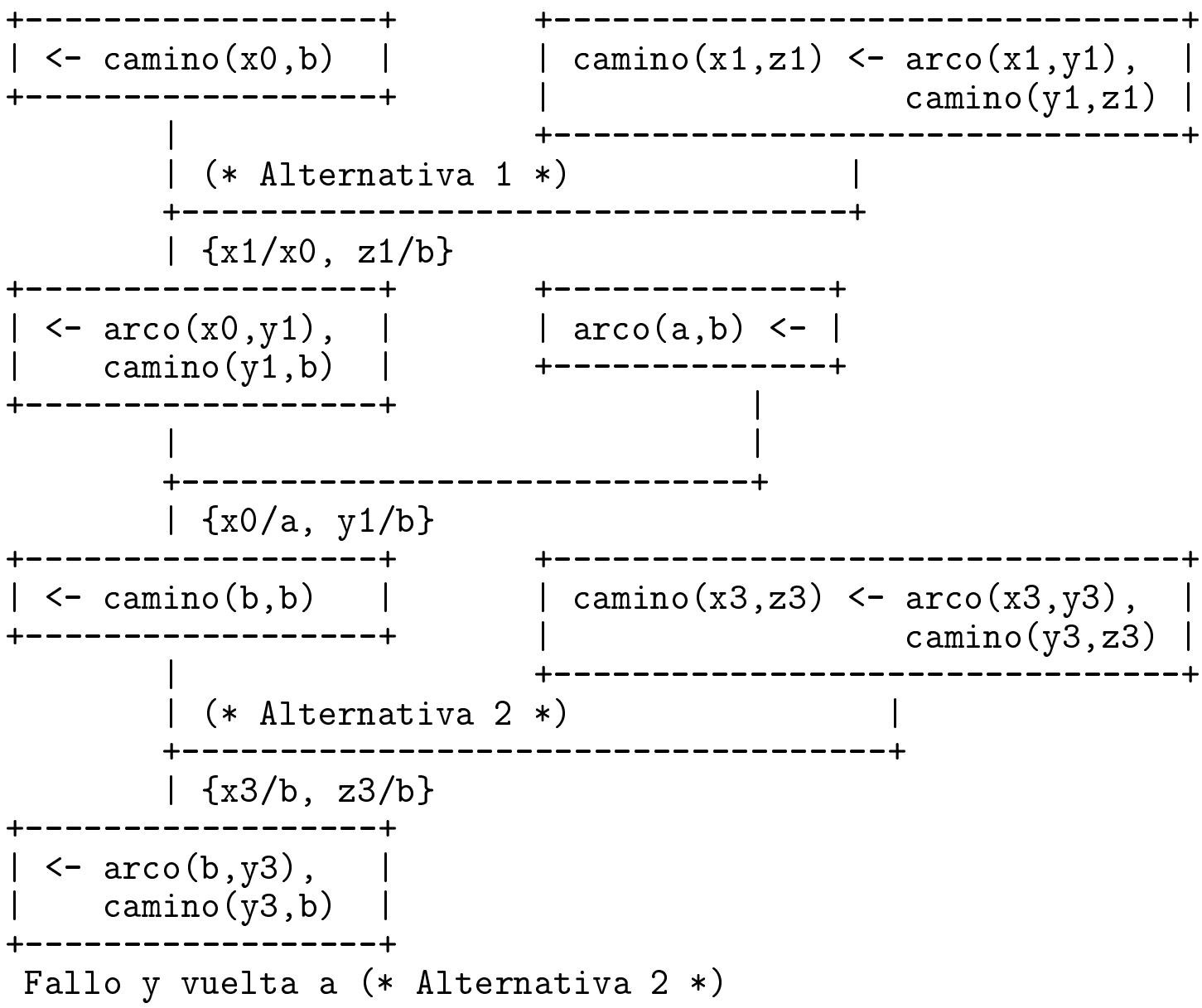
- **Ejemplo de programa lógico**

```
camino(x,z) <- arco(x,y), camino(y,z)
camino(x,x) <-
arco(a,b)   <-
```

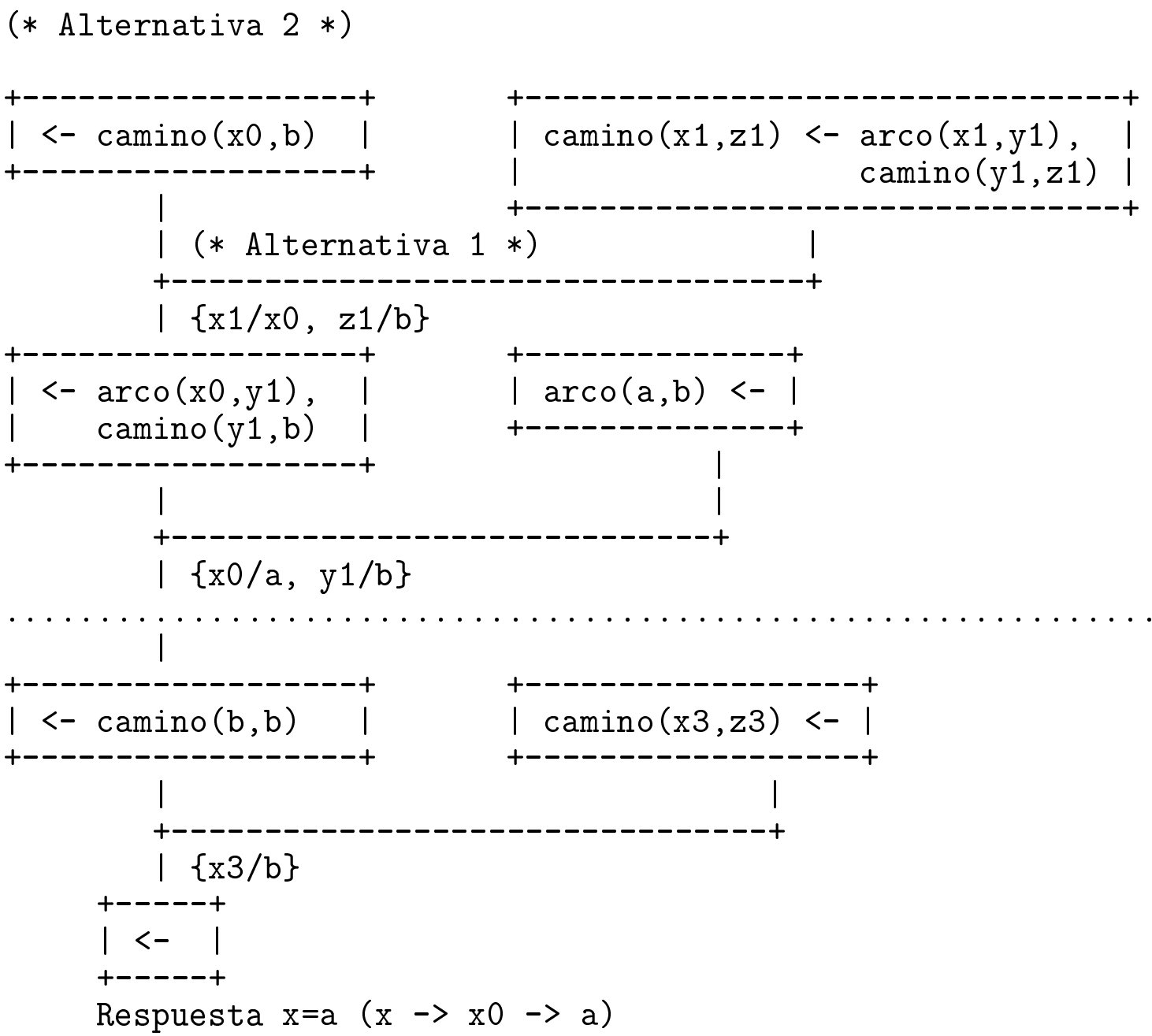
- **Ejemplo de pregunta**

```
<- camino(x,b)
```

Computación de respuestas

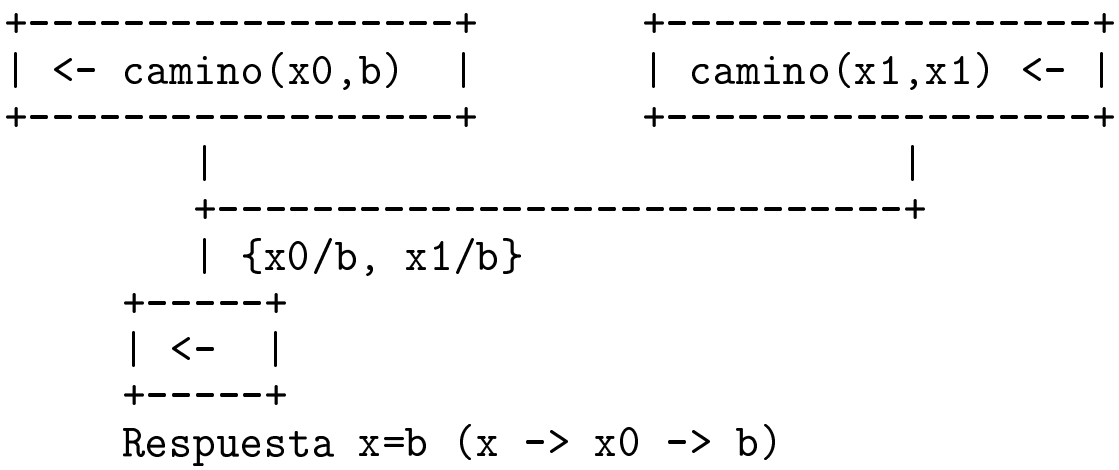


Computación de respuestas



Computación de respuestas

```
(* Alternativa 1 *)
```



Respuestas mediante entornos

- **Programa y pregunta**

```
p(x,z) <- q(x,y), p(y,z)
p(x,x) <-
q(a,b) <-

<- p(x,b)
```

- **Computación de respuestas mediante entornos**

```
[p(x,b)]0
E0 = {}
  |                              p(x,z) <- q(x,y), p(y,z)
  |                              (* Alternativa 1 *)
[q(x,y),p(y,z)]1, []0
E1 = E0 U {x0/x1, z1/b0}
  |                              q(a,b) <-
[]2, [p(y,z)]1, []0
E2 = E1 U {x1/a2, y1/b2}
  |
[p(y,z)]1, []0
  |                              p(x,z) <- q(x,y), p(y,z)
  |                              (* Alternativa 2 *)
[q(x,y),p(y,z)]3, []1, []0
E3 = E2 U {x3/b2, z3/b0}
  |
Fallo y vuelta a (* Alternativa 2 *)
```

Respuestas mediante entornos

```
(* Alternativa 2 *)

[p(x,b)]0
E0 = {}
  |                              p(x,z) <- q(x,y), p(y,z)
  |                              (* Alternativa 1 *)
[q(x,y),p(y,z)]1, []0
E1 = E0 U {x0/x1, z1/b0}
  |                              q(a,b) <-
[]2, [p(y,z)]1, []0
E2 = E1 U {x1/a2, y1/b2}
  |
[p(y,z)]1, []0
  |
.....................................................
  |                               p(x,x) <-
[]3, []1, []0
E3 = E2 U {x3/b2}
  |
[]1, []0
  |
[]0
  |
Respuesta x=a (x -> x0 -> x1 -> a2)
```

Respuestas mediante entornos

```
(* Alternativa 1 *)

[p(x,b)]0
E0 = {}
  |                                p(x,x) <-
[]1, []0
E1 = E0 U {x0/b0, x1/b0}
  |
[]0
  |
Respuesta x=b (x -> x0 -> b0)
```

Entornos

- $\omega = \{< n_i, x_i > / < m_i, t_i >: 1 \leq i \leq p\}$
- **Variables anotadas:** $< n_i, x_i >$
- **Términos anotados:** $< n_i, t_i >$
- **Números de nivel:** n_i
- **Nombre de variables:** x_i
- **Ligaduras:** $< n_i, x_i > / < m_i, t_i >$

Implementación de la resolución SLD

- **Representación de cláusulas de Horn**

```
Cláusula:       p(x,z) <- q(x,y), p(y,z)
Representación: ((p x z) (q x y) (p y z))
```

- **Representación de conjuntos de cláusulas de Horn**

```
p(x,z) <- q(x,y), p(y,z)
p(x,x) <-
q(a,b) <-

(setf *conjunto-de-clausulas*
      '(((p x z) (q x y) (p y z))
        ((p x x))
        ((q a b))))
```

- **Representación de listas de preguntas**

```
Pregunta:       <- p(a,x), q(x,b)
Representación: ( ((p a x) (q x b)) )

Cláusula:       ((p x z) (q x y) (p y z))
Resolvente:     ( ((q x y) (p y z)) ((q x b)) )
```

Implementación de la resolución SLD

- **Definición de variable**

```
(defun es-variable (expresion)
  (member expresion '(x y z u v w)))
```

- **Expresiones anotadas**

```
<término-anotado> ::= (<número-natural> <término>)

(nombre '(1 (f x)))  =>  (F X)
(defun nombre (termino-anotado)
  (second termino-anotado))

(es-variable-anotada '(3 y))  =>  (Y Z U V W)
(es-variable-anotada '(3 a))  =>  NIL
(defun es-variable-anotada (termino-anotado)
  (es-variable (nombre termino-anotado)))

(es-simbolo-anotado '(3 f))      =>  T
(es-simbolo-anotado '(3 (f x)))  =>  NIL
(defun es-simbolo-anotado (termino-anotado)
  (atom (nombre termino-anotado)))

<lista-anotada> ::=
  (<número> <átomo>) |
  (<número> (<expresión-1> ... <expresión-n>))
```

Implementación de la resolución SLD

```
(primera-expresion '(1 (f (g x) (h y))))  =>  (1 F)
(primera-expresion '(1 ((g x) (h y))))    =>  (1 (G X))
(primera-expresion '(1 ()))               =>  (1 NIL)
(defun primera-expresion (lista-anotada)
  (list (first lista-anotada)
        (first (second lista-anotada))))

  > (restantes-expresiones '(1 (f (g x) (h y))))
  (1 ((G X) (H Y)))
  > (restantes-expresiones '(1 ((g x) (h y))))
  (1 ((H Y)))
  > (restantes-expresiones '(1 ((h y))))
  (1 NIL)
(defun restantes-expresiones (lista-anotada)
  (list (first lista-anotada)
        (rest (second lista-anotada))))
```

- **Objetivos**

```
> (primer-atomo '(((p a x) (q x b)) ((q y a))))
(P A X)
(defun primer-atomo (lista-objetivos)
  (first (first lista-objetivos)))

> (restantes-objetivos '(((p a x) (q x b)) ((q y a))))
(((Q X B)) ((Q Y A)))
(defun restantes-objetivos (lista-objetivos)
  (cons (rest (first lista-objetivos))
        (rest lista-objetivos)))
```

Implementación de la resolución SLD

- **Entornos**

```
Representación de entornos:
  {<n1,x1>/<m1,t1>,...,<np,xp>/<mp,tp>}
  (((n1 x1) . (m1 t1)) ... ((np xp) . (mp tp)))

(termino-anotado '(0 x) '(((0 x) . (1 a)))) => (1 A)
(defun termino-anotado (variable-anotada entorno)
  (rest (assoc variable-anotada entorno :test #'equal)))

> (annade-ligadura '(1 x) '(2 b) '(((0 x) . (1 a))))
(((1 X) 2 B) ((0 X) 1 A))
> (annade-ligadura '(1 s) '(2 b) '(((0 x) . (1 a))))
(((0 X) 1 A))
(defun annade-ligadura (expresion-anotada
                        termino-anotado
                        entorno)
  (if (es-variable-anotada expresion-anotada)
      (acons expresion-anotada termino-anotado entorno)
      entorno))

> (es-soportada '(0 y) '(((1 x) 2 b) ((0 x) 1 a)))
NIL
> (es-soportada '(0 x) '(((1 x) 2 b) ((0 x) 1 a)))
(1 A)
(defun es-soportada (variable-anotada entorno)
  (termino-anotado variable-anotada entorno))
```

Implementación de la resolución SLD

```
(valor () '(((1 x) 2 b) ((0 x) 1 a)))      =>  NIL
(valor '(0 y) '(((1 x) 2 b) ((0 x) 1 a)))  =>  (0 Y)
(valor '(0 x) '(((1 x) 2 b) ((0 x) 1 a)))  =>  (1 A)
(valor '(0 x) '(((1 x) 2 b) ((0 x) 1 x)))  =>  (2 B)
(valor '(0 b) '(((1 x) 2 b) ((0 x) 1 x)))  =>  (0 B)
(defun valor (variable-anotada entorno)
  (if (not (es-soportada variable-anotada entorno))
      variable-anotada
      (valor (termino-anotado variable-anotada entorno)
             entorno)))

> (aplica '(0 (f x)) '(((0 x) 1 a)))
(F A)
> (aplica '(0 (f x y)) '(((0 x) 1 a) ((0 y) 1 (s x))
                         ((1 x) 1 b)))
(F A (S B))
> (aplica '(0 (f x z)) '(((0 x) 1 a) ((0 z) 1 (s y))))
(F A (S Y))
(defun aplica (expresion-anotada entorno)
  (cond
    ((es-variable-anotada expresion-anotada)
     (if (es-soportada expresion-anotada entorno)
         (aplica (valor expresion-anotada entorno) entorno)
         (nombre expresion-anotada)))
    ((es-simbolo-anotado expresion-anotada)
     (nombre expresion-anotada))
    (t (cons (aplica
               (primera-expresion expresion-anotada)
               entorno)
             (aplica
               (restantes-expresiones expresion-anotada)
               entorno)))))
```

Implementación de la resolución SLD

- **Unificación**

```
> (unifica '(0 x) '(1 y) '(((0 x) 1 a) ((1 y) 1 a)))
(((0 X) 1 A) ((1 Y) 1 A))
> (unifica '(0 x) '(1 y) '(((0 z) 1 a) ((1 y) 1 a)))
((0 X) 1 A) ((0 Z) 1 A) ((1 Y) 1 A))
> (unifica '(0 (P x b)) '(1 (P x z)) nil)
((((1 Z) 0 B) ((0 X) 1 X))
> (unifica '(0 b) '(2 y) '(((0 z) 1 a) ((1 y) 1 a)))
(((2 Y) 0 B) ((0 Z) 1 A) ((1 Y) 1 A))
> (unifica '(1 (P x (f x) y)) '(1 (P (g b) w z)) ())
(((1 Y) 1 Z) ((1 W) 1 (F X)) ((1 X) 1 (G B)))
```

Implementación de la resolución SLD

```
(defun unifica (expresion-1 expresion-2 entorno)
  (let ((valor-1 (valor expresion-1 entorno))
        (valor-2 (valor expresion-2 entorno)))
    (cond ((equal valor-1 valor-2) entorno)
          ((es-variable-anotada valor-1)
           (annade-ligadura valor-1 valor-2 entorno))
          ((es-variable-anotada valor-2)
           (annade-ligadura valor-2 valor-1 entorno))
          ((or (es-simbolo-anotado valor-1)
               (es-simbolo-anotado valor-2))
           (if (eq (nombre valor-1) (nombre valor-2))
               entorno
               'FALLO))
          (t (let ((nuevo-entorno
                    (unifica (primera-expresion valor-1)
                             (primera-expresion valor-2)
                             entorno)))
               (if (eq nuevo-entorno 'FALLO)
                   'FALLO
                   (unifica (restantes-expresiones valor-1)
                            (restantes-expresiones valor-2)
                            nuevo-entorno)))))))
```

Implementación de la resolución SLD

- **Obtención de respuesta por resolución**

- **Ejemplo de programa**

```
P(x,z) <- Q(x,y),P(y,z)
P(x,x)
Q(a,b)
```

- **Sesión**

```
(setf *conjunto-de-clausulas*
      '(((P x z) (Q x y) (P y z))
        ((P x x))
        ((Q a b))))
> (prueba '((P x b) (P a a)))
(((4 X) 0 A)
 ((3 X) 2 B)
 ((1 Y) 2 B)
 ((1 X) 2 A)
 ((1 Z) 0 B)
 ((0 X) 1 X))
```

Implementación de la resolución SLD

• Traza de la prueba

```
(prueba '((P x b) (P a a))) =
= (resolucion '(((P x b) (P a a)))   ; objetivos
              '(0)                   ; lista-niveles
              1                      ; nivel
              nil) =                 ; entorno
= (resolucion '(((Q x y) (P y z)) ((P a a)))
              '(1 0)
              2
              '(((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(nil ((P y z)) ((P a a)))
              '(2 1 0)
              3
              '(((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(((P y z)) ((P a a)))
              '(1 0)
              3
              '(((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(((Q x y) (P y z)) nil ((P a a)))
              '(3 1 0)
              4
              '(((3 z) 0 b) ((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a)
                ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(nil nil ((P a a)))
              '(3 1 0)
              4
              '(((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b)
                ((0 x) 1 x))) =
```

Implementación de la resolución SLD

```
= (resolucion '(nil ((P a a)))
              '(1 0)
              4
              '(((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b)
                ((0 x) 1 x))) =
= (resolucion '(((P a a)))
              '(0)
              4
              '(((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b)
                ((0 x) 1 x))) =
= (resolucion '(((Q x y) (P y z)) nil)
              '(4 0)
              5
              '(((4 z) 0 a) ((4 x) 0 a) ((3 x) 2 b) ((1 y) 2 b)
                ((1 x) 2 a) ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(nil ((P y z)) nil)
              '(5 4 0)
              6
              '(((4 y) 5 b) ((4 z) 0 a) ((4 x) 0 a) ((3 x) 2 b)
                ((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(((P y z)) nil)
              '(4 0)
              6
              '(((4 y) 5 b) ((4 z) 0 a) ((4 x) 0 a) ((3 x) 2 b)
                ((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(((Q x y) (P y z)) nil nil)
              '(6 4 0)
              7
              '(((6 z) 0 a) ((6 x) 5 b) ((4 y) 5 b) ((4 z) 0 a)
                ((4 x) 0 a) ((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a)
                ((1 z) 0 b) ((0 x) 1 x))) =
```

Implementación de la resolución SLD

```
= (resolucion '(nil nil)
              '(4 0)
              5
              '(((4 x) 0 a) ((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a)
                ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion '(nil)
              '(0)
              5
              '(((4 x) 0 a) ((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a)
                ((1 z) 0 b) ((0 x) 1 x))) =
= (resolucion nil
              nil
              5
              '(((4 x) 0 a) ((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a)
                ((1 z) 0 b) ((0 x) 1 x))) =
= (((4 x) 0 a) ((3 x) 2 b) ((1 y) 2 b) ((1 x) 2 a) ((1 z) 0 b)
   ((0 x) 1 x))
```

Implementación de la resolución SLD

```
(defun prueba (objetivos)
  (resolucion (list objetivos) '(0) 1 nil))

(defun resolucion (objetivos
                   lista-niveles
                   nivel
                   entorno)
  (cond ((null objetivos) entorno)
        ((null (first objetivos))
         (resolucion (rest objetivos)
                     (rest lista-niveles)
                     nivel
                     entorno))
        (t (resolvente objetivos
                       lista-niveles
                       nivel
                       entorno))))
```

Implementación de la resolución SLD

```
(defun resolvente (objetivos
                   lista-de-niveles
                   nivel
                   entorno
                   &optional (conjunto-de-clausulas
                              *conjunto-de-clausulas*)
                             (resultado 'fallo))
  (if (or (null conjunto-de-clausulas)
          (not (eq resultado 'fallo)))
      resultado
      (let* ((clausula (first conjunto-de-clausulas))
             (cabeza (first clausula))
             (cuerpo (rest clausula))
             (nuevo-entorno (unifica
                             (list (first lista-de-niveles)
                                   (primer-atomo objetivos))
                             (list nivel cabeza)
                             entorno)))
        (resolvente objetivos
                    lista-de-niveles
                    nivel
                    entorno
                    (rest conjunto-de-clausulas)
                    (if (eq nuevo-entorno 'fallo)
                        'fallo
                        (resolucion
                          (cons cuerpo
                                (restantes-objetivos objetivos
                          (cons nivel lista-de-niveles)
                          (1+ nivel)
                          nuevo-entorno))))))
```

Implementación de la resolución SLD

- **Escritura**

```
  > (respuesta '((P x b) (P y a)))
  Y = A
  X = A
  SI
  > (respuesta '((P b a)))
  NO
(defun respuesta (lista-objetivos)
  (let ((variables (variables lista-objetivos))
        (entorno (prueba lista-objetivos)))
       (cond ((eq entorno 'FALLO)
              (format t "~&No.~%"))
             (t (loop for x in variables do
                  (format t "~&~a = ~a"
                            x (aplica (list 0 x) entorno)))
                (format t "~&Si.~%")))))

(variables '(((P x y)))) => (X Y)
(defun variables (expresion)
  (cond ((null expresion) nil)
        ((es-variable (first expresion))
         (adjoin (first expresion)
                 (variables (rest expresion))))
        ((atom (first expresion))
         (variables (rest expresion)))
        (t (union (variables (first expresion))
                  (variables (rest expresion))))))
```

Ejemplos de programas lógicos

- **Aritmética natural**

```
> (setf *conjunto-de-clausulas*
      '(((suma 0 y y))
        ((suma (s x) y (s z))
         (suma x y z))))
(((SUMA 0 Y Y))
 ((SUMA (S X) Y (S Z))
  (SUMA X Y Z)))
> (respuesta '((suma (s 0) (s 0) x)))
X = (S (S 0))
Si.
NIL
> (respuesta '((suma x (s 0) (s (s 0)))))
X = (S 0)
Si.
NIL
```

Ejemplos de programas lógicos

```
> (setf *conjunto-de-clausulas*
        (append *conjunto-de-clausulas*
                '(((producto 0 y 0))
                  ((producto (s x) y z)
                   (producto x y u)
                   (suma u y z)))))
(((SUMA 0 Y Y))
 ((SUMA (S X) Y (S Z))
  (SUMA X Y Z))
 ((PRODUCTO 0 Y 0))
  ((PRODUCTO (S X) Y Z)
   (PRODUCTO X Y U)
   (SUMA U Y Z)))
> (respuesta '((producto (s (s 0)) (s (s (s 0))) x)))
X = (S (S (S (S (S (S 0))))))
Si.
NIL
> (respuesta '((producto (s (s 0)) x (s (s (s (s (s (s 0)))
X = (S (S (S 0)))
Si.
NIL
```

Ejemplos de programas lógicos

```
> (setf *conjunto-de-clausulas*
        (append *conjunto-de-clausulas*
                '(((factorial 0 (s 0)))
                  ((factorial (s x) y)
                   (factorial x z)
                   (producto (s x) z y)))))
(((SUMA 0 Y Y))
 ((SUMA (S X) Y (S Z))
  (SUMA X Y Z))
 ((PRODUCTO 0 Y 0))
  ((PRODUCTO (S X) Y Z)
   (PRODUCTO X Y U)
   (SUMA U Y Z))
  ((FACTORIAL 0 (S 0)))
  ((FACTORIAL (S X) Y)
   (FACTORIAL X Z)
   (PRODUCTO (S X) Z Y)))
> (respuesta '((factorial (s (s (s 0))) x)))
X = (S (S (S (S (S (S 0))))))
Si.
NIL
```

Ejemplos de programas lógicos

- **Concatenación de listas**

```
> (setf *conjunto-de-clausulas*
      '(((append nil x x))
        ((append (cons x y) z (cons x u))
         (append y z u))))
(((APPEND NIL X X))
 ((APPEND (CONS X Y) Z (CONS X U))
  (APPEND Y Z U)))
> (respuesta '((append (cons a (cons b nil)) (cons c nil) z
Z = (CONS A (CONS B (CONS C NIL)))
Si.
NIL
> (respuesta '((append x (cons b nil) (cons a (cons b nil))
X = (CONS A NIL)
Si.
NIL
> (respuesta '((append x y (cons a (cons b nil)))))
X = NIL
Y = (CONS A (CONS B NIL))
Si.
NIL
> (respuesta '((append (cons x nil) y (cons a (cons b nil))
X = A
Y = (CONS B NIL)
Si.
NIL
```