Temas de "Lógica y programación" (2004–05)

José Antonio Alonso Jiménez
Francisco Jesús Martín Mateos
José Luis Ruiz Reina

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 10 de Julio de 2005

Índice general

Capítulo 1

Introducción a la programación en Lisp

Lógica y Programación

Introducción a la programación en Lisp

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Introducción

- John McCarthy, 1958
  - List processing
  - Lambda cálculo
  - Procesamiento simbólico
- Programación aplicativa: sin efectos colaterales
- Lenguaje interpretado
  - Bucle lee–evalúa–escribe
  - Compilador
- Common Lisp
  - Clisp, GCL, Allegro, ...
- Inteligencia Artificial
- Prototipado Rápido
- Eficiencia

Primera toma de contacto

- Expresiones
  - Notación prefija: función y argumentos
  - Expresiones anidadas, paréntesis
  - Átomos y listas
  - Sintácticamente simple
- Ejemplos

```
> 1
1
> (+ 2 3)
5
> (+ (- 5 2) (* 3 3))
12
> (exit)
Bye.

Process inferior-lisp finished
```


Primera toma de contacto

- Evaluación
  - Toda expresión es evaluada
  - Evaluación de los argumentos de izquierda a derecha y de dentro hacia fuera
  - La función **`quote`** evita la evaluación de su argumento
- Ejemplos:

```
> (+ (- 5 2) (* 3 3))
12
> (quote (+ (- 5 2) (* 3 3)))
(+ (- 5 2) (* 3 3))
> '(+ (- 5 2) (* 3 3))
(+ (- 5 2) (* 3 3))
> x
*** - EVAL: variable X has no value
1. Break> abort
> 'x
X
```

Primera toma de contacto

- Algunos tipos de datos
  - Números: `1`, `2.5`, `24`
  - Símbolos: `a`, `x`, `juan`
  - Cadenas: `"hola"`, `"adios"`
  - Listas: `(esto (es) (una lista))`
    - Lista vacía: `()`, `nil`
    - El primer elemento de una lista es interpretado como una función
- Ejemplos

```
> (esto (es) (una lista))
*** - EVAL: undefined function ESTO
1. Break> abort
> '(esto (es) (una lista))
(ESTO (ES) (UNA LISTA))
```


Primera toma de contacto

- Operaciones con listas:
  - Construcción a partir de los elementos: `list`
  - Añadir un elemento a una lista: `cons`
  - Primer elemento de una lista: `car`
  - Resto de los elementos de una lista: `cdr`
- Ejemplos

```
> (list 'a (+ 1 2) nil)
(A 3 NIL)
> (cons 'a '(b c d))
(A B C D)
> (cons 'a (cons 'b nil))
(A B)
```

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
> (car (cdr (cdr '(a b c d))))
C
```

Primera toma de contacto

- Valores de verdad
  - Falso: **`nil`**
  - Verdadero: **`t`** (cualquier cosa distinta de **`nil`**)
- Operaciones lógicas
  - La *forma especial* **`if`**
  - La función **`not`**
  - Las *macros* **`and`** y **`or`**
- Ejemplos

```
> (if (listp '(a b c)) (+ 1 2) (+ 5 6))
3
> (and (listp '(1)) 3 (+ 13 4))
17
> (or (listp '(1)) 3 (+ 13 4))
T
> (and (listp '(1)) (listp 1) t)
NIL
```


Primera toma de contacto

- Predicados
  - Comprobar el tipo de un dato: **`listp`**, **`atom`**, **`numberp`**
  - Comprobar una propiedad: **`endp`**, **`oddp`**, **`evenp`**
- Ejemplos

```
> (listp '(a b c))
T
> (listp 27)
NIL
> (null nil)
T
> (numberp 5)
T
```

Primera toma de contacto

- Funciones
  - Realizar cálculos, operaciones: `+`, `-`, `*`, `/`, ...
  - Los predicados también son funciones
  - Recursión
- Ejemplo

```
> (defun pertenece (x l)
    (if (endp l)                        ; caso base
        nil
        (if (= x (car l))               ; caso base
            l
            (pertenece x (cdr l)))))    ; recursion
PERTENECE
> (pertenece 3 '(1 4 3 5 6))
(3 5 6)
> (pertenece 8 '(1 4 3 5 6))
NIL
```

- Indentación y comentarios

Primera toma de contacto

- Funciones de entrada y salida
  - Entrada: **`read`**
  - Salida: **`format`**
- Ejemplos

```
> (format t "~a mas ~a igual a ~a. ~%" 2 3 (+ 2  3))
2 mas 3 igual a 5.
NIL
> (defun pide (frase)
    (format t "~a " frase)
    (read))
PIDE
> (pide "Su edad, por favor:")
Su edad, por favor: 23
23
```

Primera toma de contacto

- Variables locales y globales
  - Entornos locales: **`let`**, **`let*`**, ...
  - Declaración de variables globales: **`defparameter`**
  - Asignaciones: **`setf`**, ...
- Ejemplos

```
> (let ((x 1) (y 2))
    (+ x y))
3
> x
*** - EVAL: variable X has no value
> (defparameter *glob* 3)
*GLOB*
> *glob*
3
> (setf *glob* 98)
98
```


Primera toma de contacto

- Arrays
- Estructuras de datos
- Estructuras de control de flujo
- Objetos
- Gráficos

Funciones de manipulación de listas

- Listas y pares punteados
- Funciones básicas de creación
  - **(CONS X Y)**

```
> (cons 'a 'b)                    => (A . B)
> (cons 'a '(b c))                => (A B C)
> (cons 'a (cons 'b '()))         => (A B)
> (cons '(a b) '(c d))            => ((A B) C D)
```

  - **(LIST X-1 ... X-N)**

```
> (list 'a 'b 'c)                 => (A B C)
> (list '(a b) '(c d))            => ((A B) (C D))
> (list)                          => NIL
```

  - **(APPEND L-1 ... L-N)**

```
> (append '(a) '(b) '(c) '(x y))  => (A B C X Y)
> (append '(a b) '(c d))          => (A B C D)
```

  - **(REVERSE L)**

```
> (reverse '(a (b c) d))          => (D (B C) A)
```


Funciones de manipulación de listas

- Funciones de acceso a listas
  - **(FIRST L), (CAR L)**

```
> (first '(a b c))                => A
> (first ())                      => NIL
```

  - **(REST L), (CDR L)**

```
> (rest '(a b c))                 => (B C)
> (rest ())                       => NIL
```

  - **(SECOND L)**

```
> (second '(a b c d))             => B
> (second '(a))                   => NIL
```

  - **(THIRD L), (FOURTH L), (FIFTH L), ..., (TENTH L)**
  - **(NTH N L)**

```
> (nth  2 '(a b c d))             => C
```

  - **(LAST L)**

```
> (last '(a b c d))               => (D)
```

Funciones de manipulación de listas

- Otras funciones sobre listas
  - **(LENGTH L)**

```
> (length '(a (b c) d))          => 3
```

  - **(NULL X)**

```
> (null (rest '(a b)))           => NIL
> (null (rest '(a)))             => T
```


Funciones aritméticas

- **(+ X-1 ... X-N)**

```
> (+ 3 7 5)      => 15
> (+)            => 0
```

- **(- X-1 ... X-N)**

```
> (- 123 7 5)    => 111
> (- 3)          => -3
```

- **(* X-1 ... X-N)**

```
> (* 2 7 5)      => 70
> (*)            => 1
```

- **(/ X Y)**

```
> (/ 6 2)        => 3
> (/ 5 2.0)      => 2.5
```

- **(MOD X Y)**

```
> (mod 7 2)      => 1
```

- **(EXPT X Y)**

```
> (expt 2 3)     => 8
```

- **(SQRT X)**

```
> (sqrt 16)      => 4
```

- **(MAX L), (MIN L)**

```
> (max 1 3 4 6 2)     => 6
> (min 1 3 4 6 2)     => 1
```

Valores y operadores lógicos

- Valores lógicos: **NIL**, **()**, **T**.
- Operadores lógicos:
  - **(NOT X)**

```
> (not (= (+ 1 1) 2))          => NIL
> (not (= (+ 1 1) 3))          => T
```

  - **(OR E-1 ... E-N)**

```
> (or nil 2 3)                 => 2
> (or (eq 'a 'b) (eq 'a 'c))   => NIL
```

  - **(AND E-1 ... E-N)**

```
> (and 1 nil 3)                => NIL
> (and 1 2 3)                  => 3
```


Predicados aritméticos

- **(= X-1 ... X-N)**

```
> (= 2 2.0 (+ 1 1))            => T
> (= 1 2 1)                    => NIL
```

- **(> X-1 ... X-N)**

```
> (> 4 3 2 1)                  => T
> (> 4 3 3 2)                  => NIL
```

- **(>= X-1 ... X-N)**

```
> (>= 4 3 3 2)                 => T
> (>= 4 3 3 5)                 => NIL
```

- **(< X-1 ... X-N)**
- **(<= X-1 ... X-N)**

Predicados de tipos

- **(ATOM X)**

```
> (atom 3)            => T
> (atom 'hola)        => T
> (atom '(1 2 3))     => NIL
```

- **(SYMBOLP X)**

```
> (symbolp 'a)        => T
> (symbolp 3)         => NIL
```

- **(NUMBERP X)**

```
> (numberp 4)         => T
> (numberp 3.4)       => T
> (numberp '(1))      => NIL
```

- **(LISTP X)**

```
> (listp 3)           => NIL
> (listp nil)         => T
> (listp '(1 2 3))    => T
```

- **(CONSP X)**

```
> (consp '(1 . 2))    => T
> (consp nil)         => NIL
> (consp '(2 5))      => T
```


Predicados de igualdad

- **(EQ X Y)**

```
> (eq 3 3)                          => T
> (eq 3 3.0)                        => NIL
> (eq 3.0 3.0)                      => NIL
> (eq (first '(a b c)) 'a)          => T
> (eq (cons 'a '(b c)) '(a b c))    => NIL
```

- **(EQL X Y)**

```
> (eql 3.0 3.0)                     => T
> (eql (cons 'a '(b))
       (cons 'a '(b)))              => NIL
```

- **(EQUAL X Y)**

```
> (equal (cons 'a '(b))
         (cons 'a '(b)))            => T
```

Predicado de pertenencia

- **(MEMBER E L)**

```
> (member 'x '(a x b x c))          => (X B X C)
> (member 'x '(a (x) b))            => NIL
> (setf l '((a b) (c d)))           => ((A B) (C D))
> (member '(c d) l)                 => NIL
> (member 2.0 '(1 2 3))             => NIL
```

- **(MEMBER E L :TEST <PREDICADO>)**

```
> (member '(c d) l)                 => NIL
> (member '(c d) l :test #'equal)   => ((C D))
> (member 2.0 '(1 2 3))             => NIL
> (member 2.0 '(1 2 3) :test #'=)   => (2 3)
> (member 2.0 '(1 2 3) :test #'<)   => (3)
```


Condicionales

- **(IF TEST ENTONCES [EN-CASO-CONTRARIO])**

```
> (if t   1 2)                      => 1
> (if nil 1)                        => NIL
```

- **(WHEN TEST E-1 ... E-N)**

```
> (when t   1 2 3)                  => 3
> (when nil 1 2 3)                  => NIL
```

- **(UNLESS TEST E-1 ... E-N)**

```
> (unless t   1 2 3)                => NIL
> (unless nil 1 2 3)                => 3
```

- **(COND L-1 ... L-N)**

```
> (defun notas (n)
    (cond ((< n 5) 'suspenso)
          ((< n 7) 'aprobado)
          ((< n 9) 'notable)
          (t 'sobresaliente) ))     => NOTAS
> (notas 8)                         => NOTABLE
```

Definiciones recursivas

- Definición de factorial

```
> (defun factorial (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
FACTORIAL
> (factorial 3)
6
```

- Seguimiento en la ejecución de una función:
  - `(TRACE F-1 ... F-N)`
  - `(UNTRACE F-1 ... F-N)`

Definiciones recursivas

- Seguimiento en la ejecución del factorial:

```
> (trace factorial *)
(FACTORIAL *)
> (factorial 2)
1. Trace: (FACTORIAL '2)
2. Trace: (FACTORIAL '1)
3. Trace: (FACTORIAL '0)
3. Trace: FACTORIAL ==> 1
3. Trace: (* '1 '1)
3. Trace: * ==> 1
2. Trace: FACTORIAL ==> 1
2. Trace: (* '2 '1)
2. Trace: * ==> 2
1. Trace: FACTORIAL ==> 2
2
> (untrace)
(* FACTORIAL)
```

Entornos locales

- **(LET ((VAR-1 VAL-1)...(VAR-M VAL-M))**
  E-1 ... E-N)

```
(setf a 9 b 7)                      =>  7
(let ((a 2)(b 3)) (+ a b))          =>  5
(+ a b)                             =>  16
(let ((x 2)(y (+ 1 x))) (+ x y))    =>  Error
```

- **(LET* ((VAR-1 VAL-1) ... (VAR-M VAL-M))**
  E-1 ... E-N)

```
(let* ((x 2)(y (+ 1 x))) (+ x y)) =>  5
```


Recursión en Lisp

- Función **subconjunto**
  - Implementación: **(subconjunto L1 L2)**
  - Ejemplos:

```
> (subconjunto () '(s 3 e 4))
T
> (subconjunto '(4 3) '(s 3 e 4))
T
> (subconjunto '(4 a 3) '(s 3 e 4))
NIL
```

  - Código:

```
(defun subconjunto (l1 l2)
  (if (endp l1)
      t
      (and (member (first l1) l2)
           (subconjunto (rest l1) l2))))
```

Recursión en Lisp

- Función **elimina-uno**
  - Implementación: **(elimina-uno L1 L2)**
  - Ejemplos:

```
> (elimina-uno 3 '(a b c d))
(A B C D)
> (elimina-uno 'b '(a b c d))
(A C D)
> (elimina-uno 'b '(a b c b d))
(A C B D)
```

  - Código

```
(defun elimina-uno (x l)
  (cond ((endp l) l)
        ((equal x (first l)) (rest l))
        (t (cons (first l)
                 (elimina-uno x (rest l))))))
```


Recursión en Lisp

- Función **permutacion**
  - Implementación: **(permutacion L1 L2)**
  - Ejemplos:

```
> (permutacion '(x 1 3 4) '(3 2 x 4))
NIL
> (permutacion '(x 2 3 4) '(3 2 x 4))
T
> (permutacion '(x 2 3 4) '(3 2 x 4 4))
NIL
```

  - Código:

```
(defun permutacion (l1 l2)
  (cond ((endp l1) (endp l2))
        ((member (car l1) l2)
         (permutacion (rest l1)
                      (elimina-uno (car l1) l2)))
        (t nil)))
```

Recursión cruzada en Lisp

- Funciones **par-p** e **impar-p**

```
(defun par-p (n)
  (cond  ((= n 0) t)
         ((= n 1) nil)
         (t (impar-p (- n 1)))))

(defun impar-p (n)
  (cond ((= n 0) nil)
        ((= n 1) t)
        (t (par-p (- n 1)))))
```


Recursión profunda en Lisp

- Función **sustituye**
- Implementación: **(sustituye N V L)**
- Ejemplos

```
> (sustituye 'a 'b '(f (c b) ((((a (b))))) d))
(F (C A) ((((A (A))))) D)
> (sustituye 'a 'b '(k (b (k (b (k b))))))
(K (A (K (A (K A)))))
```

- Código

```
(defun sustituye (nuevo viejo expr)
  (cond ((eql expr viejo) nuevo)
        ((atom expr) expr)
        (t (cons (sustituye nuevo viejo (first expr))
                 (sustituye nuevo viejo (rest expr))))))
```

Recursión profunda en Lisp

- Función **member-prof**
- Implementación: **(member-prof X L)**
- Ejemplos:

```
> (member-prof 'x '(a (b (c (d (x))))))
T
> (member-prof 'x '(x (b (c (d e)))))
T
> (member-prof 'x '(c (b (c (d)))))
NIL
```

- Código:

```
(defun member-prof (x l)
  (cond ((atom l) nil)
        ((eql x (first l)) t)
        (t (or (member-prof x (first l))
               (member-prof x (rest l))))))
```


Recursión de cola en Lisp

- Función **longitud**, versión recursiva

```
(defun longitud (l)
  (if (endp l)
      0
      (1+ (longitud (rest l)))))
```

- Función **longitud-it**, versión recursiva de cola

```
(defun longitud-it (l &optional (acum 0))
  (if (endp l)
      acum
      (longitud-it (rest l) (1+ acum))))
```

- Argumentos opcionales (**&optional**)

Recursión de cola en Lisp

- Función **n-primeros**, versión recursiva

```
(defun n-primeros (n l)
  (if (= n 0)
      ()
      (cons (car l) (n-primeros (- n 1) (cdr l)))))
```

- Función **n-primeros-it**, versión recursiva de cola

```
(defun n-primeros-it (n l &optional (res ()))
  (if (= n 0)
      (reverse res)
      (n-primeros-it (- n 1) (rest l)
                             (cons (first l) res))))
```


Recursión de cola en Lisp

- Función **aplana**, versión recursiva

```
(defun aplana (l)
  (cond ((endp l) l)
        ((consp (first l))
         (append (aplana (first l))
                 (aplana (rest l))))
        (t (cons (first l) (aplana (rest l))))))
```

- Función **aplana-it**, versión recursiva de cola

```
(defun aplana-it (l &optional (res ()))
  (cond ((endp l) (reverse res))
        ((consp (first l))
         (aplana-it (append (first l) (rest l)) res))
        (t (aplana-it (rest l) (cons (first l) res)))))
```

Bucles con LOOP

- Estructura: `(LOOP <INICIAL> <CENTRAL> <FINAL>)`
- Opciones iniciales:

```
(LOOP FOR <VARIABLE> FROM <INICIO> TO <FIN> ...)
(LOOP FOR <VARIABLE> FROM <INICIO> TO <FIN> BY <INCREMENTO> ...)
(LOOP FOR <VARIABLE> IN <LISTA> ...)
(LOOP WHILE <CONDICION> ...)
(LOOP UNTIL <CONDICION> ...)
```

- Opciones centrales:

```
(LOOP ... WHEN <CONDICION> ...)
```

- Opciones finales:

```
(LOOP ... DO <EXPRESION>)
(LOOP ... COLLECT <EXPRESION>)
(LOOP ... APPEND <EXPRESION>)
(LOOP ... THEREIS <EXPRESION>)
(LOOP ... ALWAYS <EXPRESION>)
(LOOP ... COUNT <EXPRESION>)
(LOOP ... SUMMING <EXPRESION>)
```


Bucles con LOOP

- Ejemplos:

```
> (loop for x from 1 to 7
        collect x)
(1 2 3 4 5 6 7)
> (loop for x from 1 to 7
        when (evenp x)
        count x)
3
> (loop for x from 1 to 7
        when (evenp x)
        summing (* x x))
56
> (loop for x from 1 to 7 by 2
        collect (* x x))
(1 9 25 49)
> (loop for x in '(1 3 5)
        summing x)
9
```

Bucles con LOOP

- Ejemplos:

```
> (let ((res nil))
    (loop for x from 1 to 7 do
          (setf res (cons x res)))
    res)
(7 6 5 4 3 2 1)
> (let ((x 3) (res nil))
    (loop while (> x 0) do
          (setf res (cons x res)
                x (- x 1)))
  res)
(1 2 3)
> (let ((x 3) (res nil))
    (loop until (<= x 0) do
          (setf res (cons x res)
                x (- x 1)))
    res)
(1 2 3)
```


Bucles con LOOP

- Ejemplos:

```
> (loop for i from 0 to 6 append (list i))
(0 1 2 3 4 5 6)
> (loop for i from 0 to 2 collect
        (loop for j from 0 to i collect
              (list i j)))
(((0 0)) ((1 0) (1 1)) ((2 0) (2 1) (2 2)))
> (loop for i from 0 to 2 append
        (loop for j from 0 to i collect
              (list i j)))
((0 0) (1 0) (1 1) (2 0) (2 1) (2 2))
> (loop for i from 0 to 2 append
        (loop for j from 0 to i append
              (list i j)))
(0 0 1 0 1 1 2 0 2 1 2 2)
> (loop for i from 0 to 2 collect
        (loop for j from 0 to i append
              (list i j)))
((0 0) (1 0 1 1) (2 0 2 1 2 2))
```

Bucles con LOOP

- Ejemplos:

```
> (loop for i from 1 to 10 always
        (numberp i))
T
> (loop for i from 1 to 10 thereis
        (evenp i))
T
> (loop for i from 1 to 10 thereis
        (when (evenp i) i))
2
```


Matrices en Lisp

- Creación:

```
(MAKE-ARRAY DIMENSIONES)
```

- Ejemplos

```
> (make-array '(2 2))
#2A((NIL NIL) (NIL NIL))
> (make-array '(2 1))
#2A((NIL) (NIL))
> (make-array '(1 2))
#2A((NIL NIL))
> (make-array '(2 2 1))
#3A(((NIL) (NIL)) ((NIL) (NIL)))
> (make-array '(2 1 2))
#3A(((NIL NIL)) ((NIL NIL)))
```

Matrices en Lisp

- Inicialización:

```
(MAKE-ARRAY DIMENSIONES :INITIAL-ELEMENT ELEMENTO)
(MAKE-ARRAY DIMENSIONES :INITIAL-CONTENTS EXPRESION)
```

- Ejemplos

```
> (make-array  '(2 2) :initial-element 2)
#2A((2 2) (2 2))
> (make-array '(3 2)
              :initial-contents '((a b c)
                                  (1 2 3)
                                  (x y z)))
#2A((A B C) (1 2 3) (X Y Z))
> (setf *matriz*
    (make-array '(2 3)
                :initial-contents '((a b c)
                                    (1 2 3))))
#2A((A B C) (1 2 3))
```


Matrices en Lisp

- Acceso:

```
(AREF MATRIZ INDICE-1 ... INDICE-N)
```

- Ejemplos:

```
> *matriz*
#2A((A B C) (1 2 3))
> (aref *matriz* 0 0)
A
> (aref *matriz* 1 1)
2
```

Matrices en Lisp

- Modificación:

```
(SETF (AREF MATRIZ INDICE-1 ... INDICE-N) EXPRESION)
```

- Ejemplos:

```
> *matriz*
#2A((A B C) (1 2 3))
> (setf (aref *matriz* 1 2) 'h)
H
> *matriz*
#2A((A B C) (1 2 H))
```


Matrices en Lisp

- Otras funciones:

```
> (array-dimensions *matriz*)
(2 3)
> (array-dimension *matriz* 0)
2
> (array-dimension *matriz* 1)
3
> (array-total-size *matriz*)
6
> (arrayp *matriz*)
T
```

Escritura

- **(FORMAT DESTINO CADENA-DE-CONTROL X-1 ... X-N)**
- Ejemplos

```
> (format t "~&Linea 1 ~%Linea 2")
Linea 1
Linea 2
NIL
> (format t "~&El cuadrado de ~a es ~a" 3 (* 3 3))
El cuadrado de 3 es 9
NIL
> (setf l '(a b c))
(A B C)
> (format t "~&La longitud de la lista ~a es ~a" l (length l))
La longitud de la lista (A B C) es 3
NIL
```


Escritura

- Algunas directivas:
  - **~a**: escribe el siguiente argumento
  - **~&**: comienza nueva línea, si no está al comienzo de una
  - **~%**: comienza nueva línea
  - **~Na**: escribe el siguiente argumento más los espacios suficientes para ocupar N caracteres de ancho

Lectura

- `(READ)`
- Lectura de expresiones Lisp
- Ejemplos:

```
> (setf a (read))
2                                 => 2
> a                               => 2
> (* (+ (read) (read))
3
4                                 => 7
> (read)
(+ 2 3)                           => (+ 2 3)
```


Tipo de dato función

- Datos en Lisp:
  - Números: `3`, `2.5`, `14.3`
  - Cadenas: `"hola"`, `"adios"`
  - Símbolos: `x`, `fun`, `fact`, `cuadrado`
  - Listas: `(1 s 3)`, `(1 (a (b)))`
  - Matrices, ...
- Un nuevo tipo de dato: funciones.
- Se expresan usando `lambda`

```
(lambda (x) (* x x))

(lambda (x y) (cond ((> x 0) 1)
                    ((< x 0) -1)
                    (t 0)))
```

Valores de un símbolo

- Un símbolo puede tener asignados varios valores:
  - Valor como variable: **`symbol-value`**
  - Valor como función: **`symbol-function`**
- Ejemplos

```
> (setf (symbol-value 'cuadrado) 8)
8
> (setf (symbol-function 'cuadrado)
        (lambda (x) (* x x)))
#<CLOSURE :LAMBDA (X) (* X X)>
```

- Usualmente:

```
> (setf cuadrado 8)
8
> (defun cuadrado (x) (* x x))
CUADRADO
```


Evaluación en Lisp

- Constantes que no sean listas: el valor representado: **`1, 3, "hola"`**
- Lambdas: la función representada

```
(lambda (x) (* 2 x))
```

- Símbolos (sin quote): su valor como *variable*

```
> cuadrado                    => 8
```

- Quote (**`'Exp`**): el valor representado por **`Exp`**

```
> '(cuadrado 4)               => (cuadrado 4)
```

- La función **`symbol-function`** (**`#'Exp`**): valor funcional de **`Exp`**

```
> #'cuadrado
#<CLOSURE :LAMBDA (X) (* X X)>
> #'(lambda (x) (* 2 x))
#<CLOSURE :LAMBDA (X) (* 2 X)>
```

Evaluación en Lisp

- Listas **(E1 E2 ... En)**
  - Se evalúa el valor funcional de **E1**:
    - Si es un símbolo: valor funcional
    - Si es una lambda-expresión: la función que representa
  - Se evalúan **E2**,...,**En** (los argumentos) recursivamente
  - Se aplica la función obtenida en primer lugar a los valores de los arguementos

```
> (cuadrado 4)
16
> (cuadrado cuadrado)
64
> ((lambda (m n) (+ m n))
   (cuadrado 2) cuadrado)
12
```


Funciones como argumentos

- **(FUNCALL FN E-1 ... E-N)**

```
> (funcall #'+ 1 2 3)
6
> (funcall #'cuadrado 4)
16
> (funcall #'(lambda (x) (* 2 x)) 3)
6
```

- **(APPLY FN ARGS)**

```
> (apply #'+ '(1 2 3))
6
> (apply #'max '(4 5 7))
7
> (apply #'(lambda (x y) (* 2 x y)) '(3 4))
24
```

Listas de asociación

- `((K1 . V1) (K2 . V2) ... (Kn . Vn))`
- `(ASSOC CLAVE LISTA-ASOCIACION)`

```
> (assoc 'a '((c d e) (a b) (b d)))
(A B)
> (assoc 'a '((c d e) (a b) (b d) (a c)))
(A B)
> (assoc 'a '((c d e) (b d)))
NIL
```

- `(ACONS CLAVE VALOR LISTA-ASOCIACION)`

```
> (acons 'd 'e '((c d e) (a b) (b d)))
((D . E) (C D E) (A B) (B D))
> (acons 'a 'e '((c d e) (a b) (b d)))
((A . E) (C D E) (A B) (B D))
```


Programación de segundo orden

- Ejemplo: base de datos de libros

```
(((AUTOR (P.R. TANIMOTO)) (CLASIFICACION (IA LISP))
  (TITULO (THE ELEMENTS OF ARTIFICIAL INTELLIGENCE)))
 ((AUTOR (H. WERTZ)) (CLASIFICACION (LISP))
  (TITULO (LISP (UNA INTRODUCCION A LA PROGRAMACION))))
 ((AUTOR (P.H. WINSTON)) (CLASIFICACION (IA))
  (TITULO (ARTIFICIAL INTELLIGENCE))))
```

- Funciones de acceso a la información

```
(defun autor (libro)
  (second (assoc 'autor libro)))

(defun titulo (libro)
  (second (assoc 'titulo libro)))

(defun clasificacion (libro)
  (second (assoc 'clasificacion libro)))
```

Programación de segundo orden

- Autores de la base de datos, versión recursiva

```
(defun autores-rec (libros &optional autores)
  (if (endp libros)
      (reverse autores)
      (autores-rec (rest libros)
                   (cons (autor (first libros)) autores))))
```

- **(MAPCAR PROCEDIMIENTO LISTA)**

```
> (mapcar #'cuadrado '(3 5 9))
(9 25 81)
```

- **(MAPCAR PROCEDIMIENTO LISTA-1 ... LISTA-N)**

```
> (mapcar #'+ '(1 2 3) '(4 5) '(6 7 8))
(11 14)
```

- Autores de la base de datos, versión con **mapcar**

```
(defun autores-apl (libros)
  (mapcar #'autor libros))
```


Programación de segundo orden

- Libros de IA, versión recursiva

```
(defun libros-de-ia-rec (libros &optional res)
  (cond ((endp libros) (reverse res))
        ((member 'ia (clasificacion (first libros)))
         (libros-de-ia-rec (rest libros)
                           (cons (first libros) res)))
        (t (libros-de-ia-rec (rest libros) res))))
```

- **(REMOVE-IF-NOT PROCEDIMIENTO LISTA)**

```
> (remove-if-not #'atom '((a) b (c) d))
(B C)
```

- **(REMOVE-IF PROCEDIMIENTO LISTA)**
- Libros de IA, versión con **remove-if-not**

```
(defun libros-de-ia-apl (libros)
  (remove-if-not (lambda (l) (member 'ia (clasificacion l)))
                 *base-de-libros*))
```

Programación de segundo orden

- Libros de una materia, versión recursiva

```
(defun libros-de-rec (materia libros &optional res)
  (cond ((endp libros) (reverse res))
        ((member materia (clasificacion (first libros)))
         (libros-de-rec materia
                        (rest libros)
                        (cons (first libros) res)))
        (t (libros-de-rec materia (rest libros) res))))
```

- Libros de una materia, versión con **remove-if-not**

```
(defun libros-de-apl (materia libros)
  (remove-if-not (lambda (x) (member materia (clasificacion x)))
                 libros))
```


Programación de segundo orden

- Número de libros de una materia, versión recursiva

```
(defun cuenta-libros-de-rec (materia libros &optional (res 0))
  (cond ((endp libros) res)
        ((member materia (clasificacion (first libros)))
         (cuenta-libros-de-ia-rec (rest libros) (+ 1 res)))
        (t (cuenta-libros-de-ia-rec (rest libros) res))))
```

- **(COUNT-IF PROCEDIMIENTO LISTA)**

```
> (count-if #'atom '((a) b (c) d e))
3
```

- **(COUNT-IF-NOT PROCEDIMIENTO LISTA)**
- Número de libros de una materia, versión con **count-if**

```
(defun cuenta-libros-de-apl (materia libros)
  (count-if (lambda (x) (member materia (clasificacion x)))
            libros))
```

Programación de segundo orden

- Un libro de una materia, versión recursiva

```
(defun el-primer-libro-de-rec (materia libros)
  (cond ((endp libros) nil)
        ((member materia (clasificacion (first libros)))
         (first libros))
        (t (el-primer-libro-de-rec materia (rest libros)))))
```

- **(FIND-IF PROCEDIMIENTO LISTA)**

```
> (find-if #'atom '((a) b (c) d e))
B
```

- **(FIND-IF-NOT PROCEDIMIENTO LISTA)**
- Un libro de una materia, versión con **find-if**

```
(defun el-primer-libro-de-apl (materia libros)
  (find-if (lambda (x) (member materia (clasificacion x)))
           libros))
```


Programación de segundo orden

- Todas las materias, versión recursiva

```
(defun materias-rec (libros &optional res)
  (if (endp libros)
      res
      (materias-rec (rest libros)
                    (union (clasificacion (first libros)) res))))
```

- Todas las materias, versión con **apply**, **mapcar** y **remove-duplicates**

```
(defun materias-apl (libros)
  (remove-duplicates
    (apply #'append (mapcar #'clasificacion libros))))
```

Programación de segundo orden

- ¿Son todos los libros de una materia dada?, versión recursiva

```
(defun todos-son-de-rec (materia libros)
  (if (endp libros)
      t
      (and (member materia (clasificacion (first libros)))
           (todos-son-de-rec materia (rest libros)))))
```

- **(EVERY PREDICADO LISTA)**

```
> (every #'atom '(1 a))          => T
> (every #'atom '((1) a))        => NIL
```

- **(EVERY PREDICADO LISTA-1 ... LISTA-N)**

```
> (every #'<= '(1 2) '(1 3))     => T
> (every #'< '(1 2) '(1 3))      => NIL
```

- ¿Son todos de una materia dada?, versión con **every**

```
(defun todos-son-de-apl (materia libros)
  (every (lambda (x) (member materia (clasificacion x)))
         libros))
```


Programación de segundo orden

- ¿Hay algún libro de una materia dada?, versión recursiva

```
(defun algunos-son-de-rec (materia libros)
  (if (endp libros)
      nil
      (or (member materia (clasificacion (first libros)))
          (algunos-son-de-rec materia (rest libros)))))
```

- **(SOME PREDICADO LISTA)**

```
> (some #'atom '((1) a))         => T
> (some #'atom '((1) (a)))       => NIL
```

- **(SOME PREDICADO LISTA-1 ... LISTA-N)**

```
> (some #'< '(1 2) '(1 3))       => T
> (some #'< '(1 2) '(1 0))       => NIL
```

- ¿Hay alguno de una materia dada?, versión con **some**

```
(defun algunos-son-de-apl (materia libros)
  (some (lambda (x) (member materia (clasificacion x)))
        libros))
```

Estructuras

- `(DEFSTRUCT (NOMBRE (:CONSTRUCTOR F-CONSTRUCTURA)`
  `(:CONC-NAME PREFIJO-)`
  `(:PRINT-FUNCTION F-ESCRITURA))`
  `CAMPO-1 ... CAMPO-N)`
- Ejemplo (puntos en el plano):

```
> (setf *punto-2* (copy-punto *punto-1*))
Punto de abcisa 2 y ordenada 5
(defstruct (punto (:constructor crea-punto)
                  (:conc-name coordenada-)
                  (:print-function escribe-punto)
   x
   y)
```

- Ejemplo (función de escritura):

```
(defun escribe-punto (punto &optional (canal t) profundidad)
  (format canal "Punto de abcisa ~a y ordenada ~a"
          (coordenada-x punto) (coordenada-y punto)))
```


Estructuras

- Función constructora:
  `(F-CONSTRUCTORA :CAMPO-1 VAL-1 ... :CAMPO-N VAL-N)`

```
> (setf *punto-1* (crea-punto :x 2 :y 3))
Punto de abcisa 2 y ordenada 3
```

- Funciones de acceso: `(PREFIJO-CAMPO-K ESTRUCTURA)`

```
> (coordenada-y *punto-1*)
3
> (setf (coordenada-y *punto-1*) 5)
5
> *punto-1*
Punto de abcisa 2 y ordenada 5
```

- Función reconocedora: `(NOMBRE-P EXP)`

```
> (punto-p *punto-1*)
T
> (punto-p '(2 5))
NIL
```

Estructuras

- Función de copia: **(COPY-NOMBRE ESTRUCTURA)**

```
> (setf *punto-2* (copy-punto *punto-1*))
Punto de abcisa 2 y ordenada 5
```

- Comparación: **(EQUALP EXP-1 EXP-2)**

```
> (setf *punto-3* *punto-1*)
Punto de abcisa 2 y ordenada 5
> (eq *punto-1* *punto-3*)
T
> (eq *punto-1* *punto-2*)
NIL
> (equalp *punto-1* *punto-2*)
T
```


Bibliografía

- Steele, G.L.
  Common Lisp the Language, 2nd edition (D. P., 1990).
- Winston, P.R. y Horn, B.K.
  LISP (3a. ed.) (Addison–Wesley, 1991).
- Graham, P.
  ANSI Common Lisp (Prentice-Hall, 1996).
- Lamkins, D.P.
  Successful Lisp: How to Understand and Use Common Lisp
  http://www.psg.com/~dlamkins/sl/cover.html

Capítulo 2

Sintaxis y semántica de la lógica proposicional

Lógica y Programación

Sintaxis y semántica de la lógica proposicional

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Aplicaciones de la lógica proposicional

- Planificación: manufactura automática, robótica, programación de vuelos en aerolíneas, uso de frecuencias radiales, . . .
- Planificación de procesos computacionales: uso de memoria, procesadores, dispositivos, . . .
- Diseño de Circuitos Integrados (VLSI)
- Control de multiacceso en redes Ethernet e Inalambricas.
- Reconocimiento de patrones y de voz.
- Bases de Datos Distribuidas.

Elementos de una lógica

- Elementos de una lógica:
  - Sintaxis: ¿qué expresiones son fórmulas?
  - Semántica: ¿qué significa que una fórmula F es consecuencia de un conjunto de fórmulas $\mathcal{S}$?: $\mathcal{S} \models F$
  - Cálculo: ¿qué significa que una fórmula F puede deducirse a partir de un conjunto de fórmulas $\mathcal{S}$?: $\mathcal{S} \vdash F$
- Propiedades:
  - Potencia expresiva
  - Adecuación: $\mathcal{S} \vdash F \Longrightarrow \mathcal{S} \models F$
  - Completitud: $\mathcal{S} \models F \Longrightarrow \mathcal{S} \vdash F$
  - Decidibilidad
  - Complejidad

Sintaxis de la lógica proposicional

- Alfabeto proposicional:
  - Símbolos proposicionales:
  - Conectivas lógicas: Negación: $\neg$ Condicional: $\rightarrow$ Conjunción: $\wedge$ Equivalencia: $\leftrightarrow$ Disyunción: $\vee$
  - Símbolos auxiliares: (y).
- Fórmulas proposicionales:
  - Fórmulas atómicas: símbolos proposicionales
  - $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.
- Ejemplos:
  - $((p \rightarrow q) \vee (q \rightarrow p))$
  - $(p \rightarrow (q \vee (\neg r)))$

Sintaxis de la lógica proposicional

- Metavariables:
  - SP: conjunto de símbolos proposicionales
  - PROP: conjunto de fórmulas proposicionales
  - Símbolos proposicionales: $p, p_0, p_1, \ldots, q, q_0, q_1, \ldots$
  - Fórmulas proposicionales: $F, F_0, F_1, \ldots, G, G_0, G_1, \ldots$
- Eliminación de paréntesis:
  - Eliminación de paréntesis externos: $(p \wedge q) \Longrightarrow p \wedge q$
  - Precedencia: $\neg, \wedge, \vee \rightarrow, \leftrightarrow$
    $(p \vee (\neg q \wedge r)) \rightarrow (s \vee t) \Longrightarrow p \vee \neg q \wedge r \rightarrow s \vee t$
    $((p \wedge \neg q) \vee r) \rightarrow s \Longrightarrow p \wedge \neg q \vee r \rightarrow s$
  - Asociatividad: $\wedge$ y $\vee$ asocian por la derecha
    $p \wedge (q \wedge r) \Longrightarrow p \wedge q \wedge r$

Implementación

- Representación de las conectivas: `-`, `&`, `/`, `->`, `<->`
- Funciones constructoras: `negacion`,
  `conjuncion`,
  `disyuncion`,
  `implicacion`,
  `equivalencia`
- Funciones de acceso: `op`, `arg1`, `arg2`
- Funciones reconocedoras: `es-atomica`,
  `es-negacion`,
  `es-conjuncion`,
  `es-disyuncion`,
  `es-implicacion`,
  `es-equivalencia`

Símbolos de una fórmula

- Símbolos proposicionales de una fórmula:

$$\begin{aligned} sp(p) &= \{p\} \\ sp(\neg F) &= sp(F) \\ sp(F \wedge G) &= sp(F) \cup sp(G) \\ sp(F \vee G) &= sp(F) \cup sp(G) \\ sp(F \rightarrow G) &= sp(F) \cup sp(G) \\ sp(F \leftrightarrow G) &= sp(F) \cup sp(G) \end{aligned}$$

- Implementación: **`(simbolos-proposicionales-formula` F`)`**
- Ejemplos:

```
> (simbolos-proposicionales-formula '((p / q) & ((- q) / r)))
(P Q R)
> (simbolos-proposicionales-formula '((p & q) -> r))
(P Q R)
```


Semántica de la lógica proposicional

- Valores de verdad: falso, 0
  verdadero, 1
- Funciones de verdad:
  - $fv_\neg : \{0,1\} \rightarrow \{0,1\}$

$$fv_\neg(i) = \begin{cases} 1, & \text{si } i = 0; \\ 0, & \text{si } i = 1. \end{cases}$$

  - $fv_\wedge : \{0,1\}^2 \rightarrow \{0,1\}$

$$fv_\wedge(i,j) = \begin{cases} 1, & \text{si } i = j = 1; \\ 0, & \text{en otro caso.} \end{cases}$$

  - $fv_\vee : \{0,1\}^2 \rightarrow \{0,1\}$

$$fv_\vee(i,j) = \begin{cases} 0, & \text{si } i = j = 0; \\ 1, & \text{en otro caso.} \end{cases}$$

Semántica de la lógica proposicional

- Funciones de verdad:
  - $fv_{\rightarrow} : \{0,1\}^2 \rightarrow \{0,1\}$

    $$fv_{\rightarrow}(i,j) = \begin{cases} 0, & \text{si } i = 1, j = 0; \\ 1, & \text{en otro caso.} \end{cases}$$
  - $fv_{\leftrightarrow} : \{0,1\}^2 \rightarrow \{0,1\}$

    $$fv_{\leftrightarrow}(i,j) = \begin{cases} 1, & \text{si } i = j; \\ 0, & \text{en otro caso.} \end{cases}$$
- Implementación: `funcion-de-verdad-de-negacion`
  `funcion-de-verdad-de-conjuncion`
  `funcion-de-verdad-de-disyuncion`
  `funcion-de-verdad-de-implicacion`
  `funcion-de-verdad-de-equivalencia`

Semántica de la lógica proposicional

- Interpretación:
  - Una interpretación I es un conjunto de símbolos proposicionales
  - Los símbolos de I son verdaderos y los restantes falsos
- INTERPRETACIONES: Conjunto de todas las interpretaciones
- Significado: sig : PROP $\times$ INTERPRETACIONES $\rightarrow \{0,1\}$
  - $$sig(p, I) = \begin{cases} 1, & \text{si } p \text{ pertenece a } I \\ 0, & \text{en caso contrario} \end{cases}$$
  - $sig(\neg F, I) = fv_{\neg}(sig(F, I))$
  - $sig(F \wedge G, I) = fv_{\wedge}(sig(F, I), sig(G, I))$
  - $sig(F \vee G, I) = fv_{\vee}(sig(F, I), sig(G, I))$
  - $sig(F \rightarrow G, I) = fv_{\rightarrow}(sig(F, I), sig(G, I))$
  - $sig(F \leftrightarrow G, I) = fv_{\leftrightarrow}(sig(F, I), sig(G, I))$

Semántica de la lógica proposicional

- Ejemplo: $F = (p \vee q) \wedge (\neg q \vee r)$
  $I = \{p, r\}$

$$
\begin{aligned}
sig(F, I) &= sig((p \vee q) \wedge (\neg q \vee r), I) = \\
&= fv_{\wedge}(sig(p \vee q, I), sig(\neg q \vee r, I)) = \\
&= fv_{\wedge}(fv_{\vee}(sig(p, I), sig(q, I)), \\
&\qquad fv_{\vee}(sig(\neg q, I), sig(r, I))) = \\
&= fv_{\wedge}(fv_{\vee}(1, 0), fv_{\vee}(fv_{\neg}(sig(q, I)), 1)) = \\
&= fv_{\wedge}(1, fv_{\vee}(fv_{\neg}(0), 1)) = \\
&= fv_{\wedge}(1, fv_{\vee}(1, 1)) = \\
&= fv_{\wedge}(1, 1) = \\
&= 1
\end{aligned}
$$

- Implementación: **`(significado F I)`**

Semántica de la lógica proposicional

- $I = \{p, r\}$

$$
\begin{array}{lcccccc}
F = & (p & \vee & q) & \wedge & (\neg q & \vee & r) \\
& (1 & \vee & 0) & \wedge & (\neg 0 & \vee & 1) \\
& & 1 & & \wedge & (1 & \vee & 1) \\
& & 1 & & \wedge & & 1 & \\
& & & & 1 & & &
\end{array}
$$

```
> (significado '((p / q) & ((- q) / r)) '(p r))
1
```

- $J = \{r\}$

$$
\begin{array}{lcccccc}
F = & (p & \vee & q) & \wedge & (\neg q & \vee & r) \\
& (0 & \vee & 0) & \wedge & (\neg 0 & \vee & 1) \\
& & 0 & & \wedge & (1 & \vee & 1) \\
& & 0 & & \wedge & & 1 & \\
& & & & 0 & & &
\end{array}
$$

```
> (significado '((p / q) & ((- q) / r)) '(r))
0
```

Interpretaciones de una fórmula

- $interpretaciones(F) = \{I : I \subseteq sp(F)\}$ q
- Número de interpretaciones de una fórmula: $|interpretaciones(F)| = 2^{|sp(F)|}$
- Implementación: **(interpretaciones-formula** F**)**
- Ejemplo:

```
> (interpretaciones-formula '((p / q) & ((- q) / r)))
(NIL (R) (Q) (Q R) (P) (P R) (P Q) (P Q R))
```


Modelos y contramodelos

- Modelo:
  - I modelo de $F \Longleftrightarrow sig(F, I) = 1$
  - Representación: $I \models F$
  - Implementación: **(es-modelo-formula** I F**)**
  - Ejemplos:

```
> (es-modelo-formula '(p r) '((p / q) & ((- q) / r)))
T
> (es-modelo-formula '(r)   '((p / q) & ((- q) / r)))
NIL
```

- Contramodelo:
  - I contramodelo de $F \Longleftrightarrow sig(F, I) = 0$
  - Representación: $I \not\models F$
  - Ejemplo: $\{r\} \not\models (p \vee q) \wedge (\neg q \vee r)$

Modelos y contramodelos

- Modelos de una fórmula
  - $modelos(F) = \{I \in interpretaciones(F) : I \models F\}$
  - Implementación: **`(modelos-formula F)`**
  - Ejemplos:

```
> (modelos-formula '(p -> q))
(NIL (Q) (P Q))
> (modelos-formula '(p & (- p)))
NIL
> (modelos-formula '((p -> q) / (q -> p)))
(NIL (P) (Q) (Q P))
```


Tablas de verdad

- Comprobación de todas las interpretaciones

p	q	r	$(p \rightarrow q)$	$(q \rightarrow r)$	$(p \rightarrow q) \vee (q \rightarrow r)$
1	1	1	1	1	1
1	1	0	1	0	1
1	0	1	0	1	1
1	0	0	0	1	1
0	1	1	1	1	1
0	1	0	1	0	1
0	0	1	1	1	1
0	0	0	1	1	1

Validez y satisfacibilidad

- Fórmulas válidas o tautologías
  - F es válida $\Longleftrightarrow$ toda interpretación de F es modelo de F
  - Representación: $\models F$
  - Implementación: `(es-valida F)`
  - Ejemplos:

```
> (es-valida '(p -> p))
T
> (es-valida '((p -> q) / (q -> p)))
T
> (es-valida '(p -> q))
NIL
```


Validez y satisfacibilidad

- Fórmulas satisfacibles
  - F es satisfacible $\Longleftrightarrow$ F tiene algún modelo
  - F es insatisfacible $\Longleftrightarrow$ F no tiene ningún modelo
  - Implementación: `(es-satisfacible F)`
    `(es-insatisfacible F)`
  - Ejemplos:

```
> (es-satisfacible '((p -> q) & (q -> r)))
T
> (es-satisfacible '(p & (- p)))
NIL
> (es-insatisfacible '(p & (- p)))
T
> (es-insatisfacible '((p -> q) & (q -> r)))
NIL
```

Validez y satisfacibilidad

- Problema de la satisfacibilidad: Dada F determinar si es satisfacible
- Problema de la validez: Dada F determinar si es válida
- Relaciones entre validez y satisfacibilidad:
  - F es válida $\Longleftrightarrow$ $\neg F$ es insatisfacible
  - F es válida $\Longrightarrow$ F es satisfacible
  - F es satisfacible $\not\Longrightarrow$ $\neg F$ es insatisfacible
- El problema de la satisfacibilidad es NP–completo

Conjuntos de fórmulas

- Símbolos proposicionales de un conjunto de fórmulas
  - $sp(\mathcal{S}) = \bigcup\{sp(F) : F \in \mathcal{S}\}$
  - Implementación: **`(simbolos-proposicionales-conjunto` $\mathcal{S}$`)`**
  - Ejemplo:

```
> (simbolos-proposicionales-conjunto
                        '(((p & q) -> r) (p -> s)))
(Q R P S)
```

- Interpretaciones de un conjunto de fórmulas
  - $interpretaciones(\mathcal{S}) = \{I : I \subseteq sp(\mathcal{S})\}$
  - Implementación: **`(interpretaciones-conjunto` $\mathcal{S}$`)`**
  - Ejemplo:

```
> (interpretaciones-conjunto '((p -> q) (q -> r)))
(NIL (R) (Q) (Q R) (P) (P R) (P Q) (P Q R))
```

Conjuntos de fórmulas

- Modelo de un conjunto de fórmulas
  - I modelo de $\mathcal{S} \Longleftrightarrow$ para toda F de $\mathcal{S}$, $I \models F$
  - Implementación: `(es-modelo-conjunto I S)`
  - Ejemplos:

```
> (es-modelo-conjunto
              '(p r) '(((p / q) & ((- q) / r)) (q -> r)))
T
> (es-modelo-conjunto
              '(p r) '(((p / q) & ((- q) / r)) (r -> q)))
NIL
```

- Contramodelo de un conjunto de fórmulas
  - I contramodelo de $\mathcal{S} \Longleftrightarrow$ para alguna F de $\mathcal{S}$, $I \not\models F$
  - Representación: $I \not\models \mathcal{S}$
  - Ejemplo: $\{p, r\} \not\models \{(p \vee q) \wedge (\neg q \vee r), r \rightarrow q\}$

Conjuntos de fórmulas

- Modelos de un conjunto de fórmulas
  - $modelos(\mathcal{S}) = \{I \in interpretaciones(\mathcal{S}) : I \models \mathcal{S}\}$
  - Implementación: `(modelos-conjunto S)`
  - Ejemplos:

```
> (modelos-conjunto '(((p / q) & ((- q) / r)) (q -> r)))
((Q R) (P) (P R) (P Q R))
> (modelos-conjunto '(((p / q) & ((- q) / r)) (r -> q)))
((R Q) (P) (P R Q))
```

Conjuntos de fórmulas

- Conjuntos consistentes e inconsistentes de fórmulas
  - $\mathcal{S}$ es consistente $\Longleftrightarrow$ $\mathcal{S}$ tiene algún modelo
  - $\mathcal{S}$ es inconsistente $\Longleftrightarrow$ $\mathcal{S}$ no tiene ningún modelo
  - Implementación: `(es-consistente` $\mathcal{S}$`)`
    `(es-inconsistente` $\mathcal{S}$`)`
  - Ejemplos:

```
> (es-consistente '((p / q) & ((- q) / r) (p -> r)))
T
> (es-consistente '((p / q) & ((- q) / r) (p -> r) (- r)))
NIL
> (es-inconsistente '((p / q) & ((- q) / r) (p -> r)))
NIL
> (es-inconsistente '((p / q) & ((- q) / r) (p -> r) (- r)))
T
```


Consecuencia lógica

- F es consecuencia de $\mathcal{S}$ $\Longleftrightarrow$ los modelos de $\mathcal{S}$ son modelos de F
- Representación: $\mathcal{S} \models F$
- Implementación: `(es-consecuencia` $\mathcal{S}$ F`)`
- Ejemplos:

```
> (es-consecuencia '((p -> q) (q -> r)) '(p -> r))
T
> (es-consecuencia '(p) '(p & q))
NIL
```

- Las siguientes condiciones son equivalentes
  - $\{F_1, \ldots, F_n\} \models G$
  - $\models F_1 \wedge \ldots \wedge F_n \rightarrow G$
  - $\{F_1, \ldots, F_n, \neg G\}$ es inconsistente

Ejercicios

- *Sean $\mathcal{S} = \{p \to (q \to \neg r), p \vee s, (\neg q \wedge r) \to s\}$ y $F = s \vee \neg r$. Se pide comprobar que $\mathcal{S} \models F$ mediante la definición de consecuencia lógica*
- *Sean $\mathcal{S} = \{(p \wedge \neg q) \to r, (\neg p \vee r) \to s, q \to \neg r\}$ y $F = r \to s$. Se pide comprobar que $\mathcal{S} \models F$ mediante la definición de consecuencia lógica*
- *Sean $\mathcal{S} = \{a \to \neg b, \neg a \to (c \vee b), b \to (d \wedge a), d \to \neg c\}$ y $F = a \vee \neg d$. Se pide comprobar que $\mathcal{S} \models F$ mediante la definición de consecuencia lógica*
- *Sean $\mathcal{S} = \{a \to \neg b, \neg b \to (c \vee d), c \to \neg d, d \to (b \wedge \neg a)\}$ y $F = b \vee \neg d$. Se pide comprobar que $\mathcal{S} \models F$ mediante la definición de consecuencia lógica*

Referencias

- Chang, C.L. y Lee, R.C.T *Symbolic Logic and Mechanical Theorem Proving*. (Academic Press, 1973)
  - Cap. 2: “The propositional logic”
- Genesereth, M.R. *Computational Logic*
  - Cap. 2: “Propositional logic”
- Nilsson, N.J. *Inteligencia artificial (Una nueva síntesis)*. (McGraw–Hill, 2000)
  - Cap. 13: “El cálculo proposicional”
- Russell, S. y Norvig, P. *Inteligencia artificial (un enfoque moderno)*. (Prentice Hall Hispanoamericana, 1996)
  - Cap. 6.4: “Lógica propositiva: un tipo de lógica muy sencillo”

Capítulo 3

Representación de problemas

Lógica y Programación

Representación de problemas

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

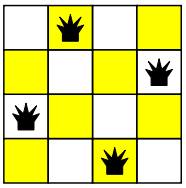
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Introducción

- Cálculo de respuestas
  - Problema de las N reinas
  - Planificación de tareas
- Comprobación de propiedades
  - Validación de circuitos lógicos
  - Recorridos imposibles

Problema de las N reinas

- Colocar N reinas en un tablero rectangular de dimensiones $N \times N$ de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal. Ejemplo para $N = 4$



- Pasos a seguir
  - Seleccionar elementos básicos del problema
  - Expresar la semántica del problema en función de los elementos básicos

Problema de las N reinas

- Elementos básicos del problema
  - Cada casilla puede estar libre u ocupada por una reina
  - Hay dos posibles situaciones: Valores de verdad
  - Una variable proposicional por cada casilla: $R_{i,j}$
    - $R_{i,j}$ es verdadera si y sólo si la casilla (i, j) está ocupada por una reina
    - $R_{i,j}$ es falsa si y sólo si la casilla (i, j) está libre
  - El estado de las variables representa una posible solución al problema: Interpretación
    - $\{R_{1,2}, R_{2,4}, R_{3,1}, R_{4,3}\}$ representa una solución **correcta** al problema para $N = 4$
    - $\{R_{1,1}, R_{2,2}, R_{3,3}, R_{4,4}\}$ representa una solución **incorrecta**
    - $\{\}$ representa una solución **incorrecta**

Problema de las N reinas

- Semántica del problema: Fórmulas
  - No debe haber dos reinas en una misma línea
    - Las casillas $(1,1)$ y $(1,2)$ no pueden estar ocupadas al mismo tiempo por dos reinas: $\neg(R_{1,1} \wedge R_{1,2})$
    - Las casillas $(1,3)$ y $(3,1)$ no pueden estar ocupadas al mismo tiempo por dos reinas: $\neg(R_{1,3} \wedge R_{3,1})$
    - Dadas dos casillas R_{i_1,j_1} y R_{i_2,j_2} dentro de una misma línea: $\neg(R_{i_1,j_1} \wedge R_{i_2,j_2})$
  - Se han de colocar N reinas en el tablero $\Longleftrightarrow$ Hay una reina en cada fila $\Longleftrightarrow$ Hay una reina en cada columna
    - Hay una reina en la primera fila: $R_{1,1} \vee R_{1,2} \vee \ldots \vee R_{1,N}$

Problema de las N reinas

- Para $N = 4$:
  - Variables: $R_{i,j}$ con $1 \leq i, j \leq 4$
  - Fórmulas: $\mathcal{S}$
    - Jaque en horizontal: $\neg(R_{i,j_1} \wedge R_{i,j_2})$, con $1 \leq i \leq 4$ y $1 \leq j_1 < j_2 \leq 4$
    - Jaque en vertical: $\neg(R_{i_1,j} \wedge R_{i_2,j})$, con $1 \leq j \leq 4$ y $1 \leq i_1 < i_2 \leq 4$
    - Jaque en diagonal: $\neg(R_{i_1,j_1} \wedge R_{i_2,j_2})$, con $1 \leq i_1 < i_2 \leq 4$, $1 \leq j_1, j_2 \leq 4$ y $|i_2 - i_1| = |j_2 - j_1|$
    - Una reina por fila: $R_{i,1} \vee R_{i,2} \vee R_{i,3} \vee R_{i,4}$, con $1 \leq i \leq 4$
  - $\mathcal{S}$ es consistente si y sólo si el problema tiene solución
  - $I \models \mathcal{S}$ si y sólo si la interpretación I representa una solución al problema

Problema de las N reinas

- Generación de los símbolos para representar las casillas

```
(defun casilla-r (i j)
  (intern (format nil "R-~a-~a" i j)))
```

- Ejemplos

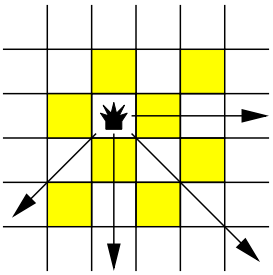
```
> (casilla-r 1 1)
R-1-1 ;
NIL
> (casilla-r 2 3)
R-2-3 ;
NIL
```

- Constante para almacenar el tamaño del tablero

```
> (defparameter *orden* 4)
*ORDEN*
```


Problema de las N reinas

- Determinar los pares de casillas que se dan jaque:



- Casilla (i, j):
  - Línea horizontal: $(i, j+1), \ldots, (i, N)$
  - Línea vertical: $(i+1, j), \ldots, (N, j)$
  - Línea diagonal principal: $(i+1, j+1), \ldots, (i+k, j+k)$, con $k = min(N-i, N-j)$
  - Línea diagonal secundaria: $(i+1, j-1), \ldots, (i+k, j-k)$, con $k = min(N-i, j-1)$

Problema de las N reinas

- Determinar los pares de casillas que se dan jaque:
  - Implementación: **(casillas** i j**)**
  - Ejemplos:

```
> (casillas 1 1)
((1 2) (1 3) (1 4)
 (2 1) (3 1) (4 1)
 (2 2) (3 3) (4 4))
> (casillas 2 3)
((2 4)
 (3 3) (4 3)
 (3 4)
 (3 2) (4 1))
```


Problema de las N reinas

- Fórmulas 'Jaque en horizontal', 'Jaque en vertical' y 'Jaque en diagonal' para una casilla dada (i_1, j_1)
  - Para cada casilla (i_2, j_2) en **(casillas** i_1 j_1**)**, generar la fórmula $\neg(R_{i_1,j_1} \wedge R_{i_2,j_2})$
- Implementación: **(sin-jaque** i_1 j_1**)**
- Ejemplos

```
> (sin-jaque 1 1)
((- (R-1-1 & R-1-2)) (- (R-1-1 & R-1-3))
 (- (R-1-1 & R-1-4)) (- (R-1-1 & R-2-1))
 (- (R-1-1 & R-3-1)) (- (R-1-1 & R-4-1))
 (- (R-1-1 & R-2-2)) (- (R-1-1 & R-3-3))
 (- (R-1-1 & R-4-4)))
> (sin-jaque 2 3)
((- (R-2-3 & R-2-4)) (- (R-2-3 & R-3-3))
 (- (R-2-3 & R-4-3)) (- (R-2-3 & R-3-4))
 (- (R-2-3 & R-3-2)) (- (R-2-3 & R-4-1)))
```

Programación de segundo orden

- Aplicar una función binaria a una lista de argumentos:

$$2^{\left(2^{\left(2^{\cdot^{\cdot^{\cdot^{2}}}}\right)}\right)}$$

- **`(REDUCE PROCEDIMIENTO LISTA :FROM-END VAL)`**

```
> (reduce #'expt '(2 2 2 2))
256
> (reduce #'expt '(2 2 2 2) :from-end t)
65536
> (reduce #'list '(2 2 2 2))
(((2 2) 2) 2)
> (reduce #'list '(2 2 2 2) :from-end t)
(2 (2 (2 2)))
```


Problema de las N reinas

- Fórmulas 'Una reina por fila', para una fila dada i
  - Generar la fórmula $R_{i,1} \vee R_{i,2} \vee \ldots \vee R_{i,N}$
- Implementación: **`(reina-en-fila` i`)`**
- Ejemplos

```
> (reina-en-fila 1)
(((R-1-1 / R-1-2) / R-1-3) / R-1-4)
> (reina-en-fila 2)
(((R-2-1 / R-2-2) / R-2-3) / R-2-4)
```

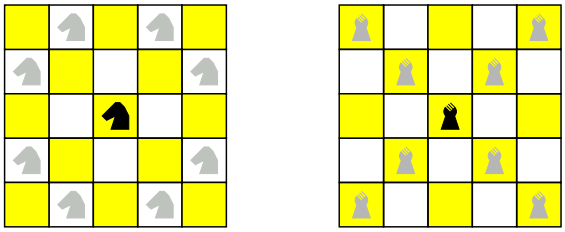
Problema de las N reinas

- Conjunto de fórmulas para un N dado
- Implementación: **`(n-reinas N)`**
- Ejemplos

```
> (n-reinas 4)
((- (R-1-1 & R-1-2)) ... (- (R-1-1 & R-4-4))
 (- (R-1-2 & R-1-3)) ... (- (R-1-2 & R-2-1))
 (- (R-1-3 & R-1-4)) ... (- (R-1-3 & R-3-1))
 (- (R-1-4 & R-2-4)) ... (- (R-1-4 & R-4-1))
 ...
 (- (R-4-1 & R-4-2)) ... (- (R-4-1 & R-4-4))
 (- (R-4-2 & R-4-3)) (- (R-4-2 & R-4-4))
 (- (R-4-3 & R-4-4))
 (((R-1-1 / R-1-2) / R-1-3) / R-1-4)
 (((R-2-1 / R-2-2) / R-2-3) / R-2-4)
 (((R-3-1 / R-3-2) / R-3-3) / R-3-4)
 (((R-4-1 / R-4-2) / R-4-3) / R-4-4))
```


Problema de los caballos y los alfiles

- Situar en un tablero de ajedrez de dimensiones $N \times N$, tantos caballos y alfiles como sea posible sin que se den jaque mútuamente.



- Se pide representar el problema en lógica proposicional y encontrar todas las soluciones para 3 filas y 3 columnas en las que aparezcan la mayor cantidad posible de figuras

Problema de los caballos y los alfiles

- Elementos básicos del problema
  - Cada casilla puede estar libre u ocupada por un caballo o un alfil
  - Hay tres posibles situaciones
  - Dos variables proposicionales por cada casilla: $C_{i,j}$ y $A_{i,j}$
    - $C_{i,j}$ es verdadera si y sólo si la casilla (i, j) está ocupada por un caballo
    - $A_{i,j}$ es verdadera si y sólo si la casilla (i, j) está ocupada por un alfil
  - El estado de las variables representa una posible solución al problema: Interpretación

Problema de los caballos y los alfiles

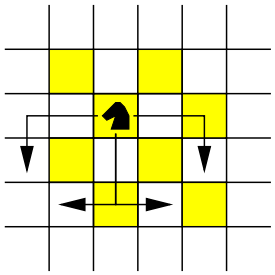
- Semántica del problema: Fórmulas
  - Las figuras no se deben dar jaque entre sí
    - Si en la casilla $(1, 1)$ hay un alfil, entonces no puede haber ninguna figura en la casilla $(2, 2)$: $A_{1,1} \rightarrow (\neg A_{2,2} \wedge \neg C_{2,2})$
    - Si en la casilla $(1, 1)$ hay un caballo, entonces no puede haber ninguna figura en la casilla $(2, 3)$: $C_{1,1} \rightarrow (\neg A_{2,3} \wedge \neg C_{2,3})$
  - Una misma casilla no puede estar simultáneamente ocupada por un alfil y un caballo. Para la casilla $(1, 1)$: $\neg(C_{1,1} \wedge A_{1,1})$

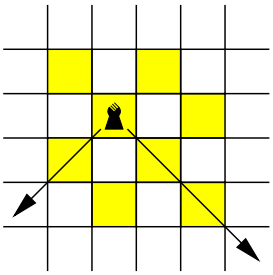
Problema de los caballos y los alfiles

- Para $N = 3$:
  - Variables: $A_{i,j}$ y $C_{i,j}$ con $1 \leq i, j \leq 3$
  - Fórmulas: $\mathcal{S}$
    - Jaque por alfil: $A_{i_1,j_1} \rightarrow (\neg C_{i_2,j_2} \wedge \neg A_{i_2,j_2})$, con $1 \leq i_1 < i_2 \leq 3$, $1 \leq j_1, j_2 \leq 3$ y $|i_2 - i_1| = |j_2 - j_1|$
    - Jaque por caballo: $C_{i_1,j_1} \rightarrow (\neg C_{i_2,j_2} \wedge \neg A_{i_2,j_2})$, con $1 \leq i_1 < i_2 \leq 3$, $1 \leq j_1, j_2 \leq 3$ y $(|i_2 - i_1|, |j_2 - j_1|) \in \{(1,2), (2,1)\}$
    - Una ficha por casilla: $\neg(A_{i,j} \wedge C_{i,j})$, con $1 \leq i, j \leq 3$
  - $I \models \mathcal{S}$ si y sólo si la interpretación I representa una solución al problema
  - Buscamos el modelo con más elementos

Problema de los caballos y los alfiles

- Determinar los pares de casillas que se dan jaque:

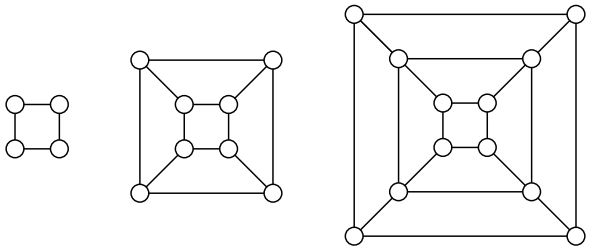




- Casilla (i, j):
  - Jaque por caballo: $(i + 1, j - 2)$, $(i + i, j + 2)$, $(i + 2, j - 1)$ y $(i + 2, j + 1)$
  - Jaque por alfil (diagonal principal): $(i + 1, j + 1), \ldots,$ $(i + k, j + k)$, con $k = min(N - i, N - j)$
  - Jaque por alfil (diagonal secundaria): $(i + 1, j - 1), \ldots,$ $(i + k, j - k)$, con $k = min(N - i, j - 1)$

Problema del coloreado de grafos

- Colorear los nodos de un grafo con K colores, de forma que no haya dos vértices adyacentes con el mismo color
- Grafos a considerar:



- Dado un número natural N, el grafo tiene N niveles y en cada nivel hay 4 nodos, $X_{i,0}$, $X_{i,1}$, $X_{i,2}$ y $X_{i,3}$. Los nodos están relacionados de la siguiente forma:
  - Para cualquier nivel i, hay arcos entre los nodos $X_{i,j}$ y $X_{i,j+1}$, con $0 \leq j \leq 2$, y otro arco entre $X_{i,3}$ y $X_{i,0}$
  - Para cualquier nivel i, con $1 < i$, hay un arco entre $X_{i,j}$ y $X_{i-1,j}$, para $j = 0, 1, 2, 3$

Problema del coloreado de grafos

- Elementos básicos del problema
  - Asignar un color a cada uno de los M nodos
  - Cada nodo puede tener uno entre K colores: hay K posibles situaciones
  - K variables proposicionales por nodo: $N_{i,j}$
    - $N_{i,j}$ es verdadera si y sólo si el nodo i-ésimo tiene asignado el color j-ésimo
  - El estado de las variables representa una posible solución al problema: Interpretación
- Enumeración de los nodos ($M = 4N$)

Nodo	Número
$X_{i,j}$	$4(i-1)+j$

Problema del coloreado de grafos

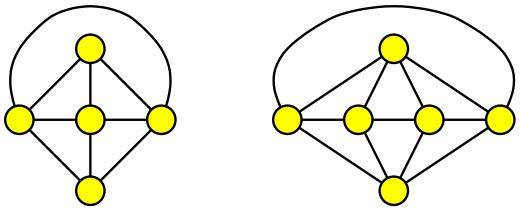
- Semántica del problema: Fórmulas
  - Dos nodos adyacentes no tienen asignado el mismo color
    - Si el tercer nodo del segundo nivel tiene asignado el color 1, entonces el segundo nodo del segundo nivel no puede tener asignado dicho color: $\neg(N_{6,1} \wedge N_{5,1})$
  - Todo nodo tiene asignado algún color
    - El tercer nodo del segundo nivel tiene asignado algún color: $N_{6,1} \vee \ldots \vee N_{6,K}$
  - Todo nodo tiene asignado un único color
    - Si el tercer nodo del segundo nivel tiene asignado el color 1, entonces no puede tener asignado ningún otro color: $N_{6,1} \longrightarrow (\neg N_{6,2} \wedge \neg N_{6,3} \wedge \ldots \wedge \neg N_{6,K})$

Problema del coloreado de grafos

- Para $N = 3$ ($M = 12$) y $K = 3$
  - Variables: $N_{h,c}$, con $0 \leq h \leq 11$, $1 \leq c \leq 3$
  - Fórmulas: $\mathcal{S}$
    - Adyacentes en el mismo nivel con distinto color: $\neg(N_{h_1,c} \wedge N_{h_2,c})$, con $1 \leq c \leq 3$, $0 \leq h_1 < h_2 \leq 11$, $h_1 = 4n + r_1$, $h_2 = 4n + r_2$, $0 \leq r_1, r_2 \leq 3$ y $r_2 - r_1 \in \{1, 3\}$
    - Adyacentes en distintos niveles con distinto color: $\neg(N_{h_1,c} \wedge N_{h_2,c})$, con $1 \leq c < 3$, $0 \leq h_1 < h_2 \leq 11$ e $h_2 = h_1 + 4$
    - Nodos con algún color: $N_{h,1} \vee N_{h,2} \vee N_{h,3}$, con $0 \leq h \leq 11$
    - Nodos con un único color: $\neg(N_{h,c_1} \wedge N_{h,c_2})$, con $0 \leq h \leq 11$ y $1 \leq c_1 < c_2 \leq 3$

Recorridos eulerianos de grafos

- Recorrer un grafo, pasando de una arista a otra adyacente, de forma que todas las aristas sean utilizadas una única vez
- Grafos a considerar



- Dado un número natural N, el grafo tiene $N+4$ nodos, dos diferenciados, X_0 y X_{N+3}, conectados con todos los demás, y el resto, X_1 a X_{N+2}, conectados entre sí en círculo:
  - X_i y X_0 están conectados para cualquier i con $1 \leq i \leq N+2$
  - X_i y X_{N+3} están conectados para cualquier i con $1 \leq i \leq N+2$
  - X_i y X_{i+1} están conectados para cualquier i con $1 \leq i < N+2$
  - X_{N+2} y X_1 están conectados

Recorridos eulerianos de grafos

- Elementos básicos del problema
  - Establecer el orden en que se han de recorrer las M aristas
  - Asignar a cada arista un valor númerico entre 1 y M
  - Por cada arista i, hay M variables: $A_{i,j}$
    - $A_{i,j}$ es verdadera si y sólo si la arista i es recorrida en el j-ésimo lugar
- Enumeración de las aristas ($M = 3N+6$)

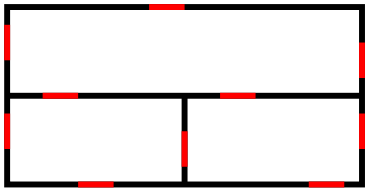
Vértices	Número de arista
X_0 y X_i, $1 \leq i \leq N+2$	i
X_{N+3} y X_i, $1 \leq i \leq N+2$	$N+2+i$
X_i y X_{i+1}, $1 \leq i < N+2$	$2N+4+i$
X_{N+2} y X_1	$3N+6$

Recorridos eulerianos de grafos

- Semántica del problema: Fórmulas
  - Todas las aristas son recorridas en algún momento
    - La arista entre los vértices X_0 y X_1 es recorrida en algún momento: $A_{1,1} \vee \ldots \vee A_{1,3N+6}$
  - Las aristas son recorridas una única vez
    - Si la arista entre los vértices X_0 y X_1 es recorrida en primer lugar, entonces no puede ser recorrida otra vez: $A_{1,1} \rightarrow (\neg A_{1,2} \wedge \neg A_{1,3} \wedge \ldots \wedge \neg A_{1,3N+6})$
  - Si una arista es recorrida en el j-ésimo lugar, con $j < 3N + 6$, entonces alguna de sus adyacentes es recorrida en el $j + 1$-ésimo lugar
    - Si la arista entre los vértices X_0 y X_1 es recorrida en primer lugar, entonces alguna de sus adyacentes es recorrida en segundo lugar: $A_{1,1} \rightarrow (A_{2,2} \vee \ldots \vee A_{N+2,2} \vee A_{2N+5,2} \vee A_{3N+6,2})$

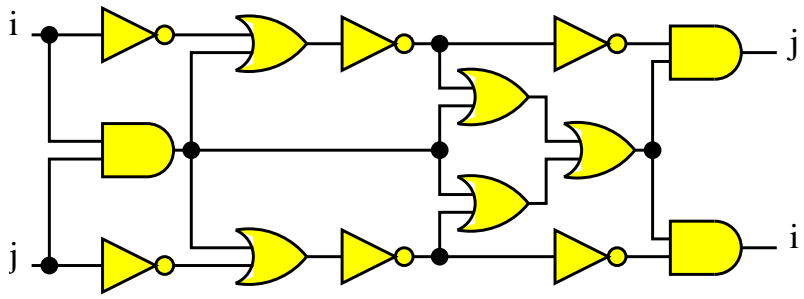
Problema de las habitaciones

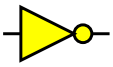
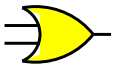
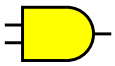
- En el recinto que se describe en la figura hay una puerta en cada uno de los muros (marcadas en color rojo). ¿Hay alguna forma de recorrer el recinto, comenzando en el exterior, pasando una y sólo una vez por cada una de las puertas y terminando en el exterior?.



Validación de circuitos lógicos

- Verificar el diseño del siguiente circuito lógico que invierte las señales de entrada



- Donde:
  - Inversor 
  - Puerta Or 
  - Puerta And 

Capítulo 4

Formas normales

Lógica y Programación

Formas Normales

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Introducción

- Simplificar las fórmulas preservando su significado
  - Reducir el número de conectivas
  - Cambiar la extructura de la fórmula
- Objetivo: Resolver fácilmente los problemas de satisfacibilidad o consecuencia lógica
- Inconvenientes:
  - Aumento del tamaño de la fórmula
  - Complejidad del proceso de normalización

Literales

- Definiciones:
  - Si F es una fórmula atómica, entonces F es un literal positivo
  - Si F es una fórmula atómica, entonces $\neg F$ es un literal negativo
  - F es un literal si y sólo si F es literal positivo o negativo
- Variables para literales: $L, L_1, L_2, \ldots$
- Implementación: `(es-literal-positivo` L`)`
  `(es-literal-negativo` L`)`
  `(es-literal` L`)`

Literales

- Complementario de un literal: $\overline{L} = \begin{cases} \neg p \text{ si } L = p \\ p \quad \text{si } L = \neg p \end{cases}$
- Implementación: `(complementario` L`)`
- Literales de una fórmula: $lit(F)$
- Implementación: `(literales-formula` F`)`
- Ejemplos

```
> (literales-formula '(p / ((- q) / r)))
(P (- Q) R)
```

Equivalencia lógica

- F y G son equivalentes si se verifica $\models F \leftrightarrow G$
- Representación: $F \equiv G$
- $F \equiv G$ si y sólo si para toda interpretación I de $\{F, G\}$ se tiene $sig(F, I) = sig(G, I)$
- Implementación: **`(es-equivalente F G)`**
- Ejemplos:

```
> (es-equivalente '(p <-> q) '((p -> q) & (q -> p)))
T
> (es-equivalente '(p -> q)  '((- p) / q))
T
> (es-equivalente '(p & q)   '(- ((- p) / (- q))))
T
> (es-equivalente '(p / q)   '(- ((- p) & (- q))))
T
```


Propiedades de la equivalencia lógica

- Las conectivas preservan la equivalencia:
  - Si $F \equiv F'$, entonces $\neg F \equiv \neg F'$
  - Si $F \equiv F'$, $G \equiv G'$ y $\star \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, entonces $F \star G \equiv F' \star G'$
- Propiedad de las subfórmulas equivalentes:
  - Sea G una subfórmula de F y F' la obtenida sustituyendo una ocurrencia de G en F por G'. Si $G \equiv G'$, entonces $F \equiv F'$

Leyes de equivalencia lógica

- Idempotencia: $F \vee F \equiv F$,
  $F \wedge F \equiv F$
- Conmutatividad: $F \vee G \equiv G \vee F$,
  $F \wedge G \equiv G \wedge F$
- Asociatividad: $F \vee (G \vee H) \equiv (F \vee G) \vee H$,
  $F \wedge (G \wedge H) \equiv (F \wedge G) \wedge H$
- Distributividad: $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$,
  $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$
- Doble negación: $\neg\neg F \equiv F$
- Leyes de De Morgan: $\neg(F \wedge G) \equiv \neg F \vee \neg G$,
  $\neg(F \vee G) \equiv \neg F \wedge \neg G$

Forma normal negativa

- Definición de forma normal negativa:
  - Si F es atómica, entonces F y $\neg F$ son formas normales negativas
  - Si F y G son formas normales negativas, entonces $(F \wedge G)$ y $(F \vee G)$ también lo son.
- Ejemplos
  - $(\neg p \vee q) \wedge (\neg q \vee p)$ es una forma normal negativa
  - $(p \rightarrow q) \wedge (q \rightarrow p)$ no es una forma normal negativa
  - $\neg(p \wedge q)$ no es una forma normal negativa

Transformación a forma normal negativa

- Objetivo: Dada una fórmula F, obtener una fórmula en forma normal negativa G tal que $F \equiv G$.
- Procedimiento $fnn(F)$
  - Eliminación de equivalencias
    $$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$$
  - Eliminación de implicaciones
    $$p \rightarrow q \equiv \neg p \vee q$$
  - Interiorización de negaciones
    $$\neg(p \wedge q) \equiv \neg p \vee \neg q$$
    $$\neg(p \vee q) \equiv \neg p \wedge \neg q$$
    $$\neg\neg p \equiv p$$
- Propiedades:
  - $fnn(F)$ es una forma normal negativa
  - $fnn(F) \equiv F$

Transformación a forma normal negativa

- Eliminación de equivalencias:
  - Implementación: **`(elimina-equivalencias F)`**
  - Ejemplos:
```
> (elimina-equivalencias '((p <-> q) & (q <-> r)))
(((P -> Q) & (Q -> P)) & ((Q -> R) & (R -> Q)))
```
- Eliminación de implicaciones:
  - Implementación: **`(elimina-implicaciones F)`**
  - Ejemplos:
```
> (elimina-implicaciones '((p -> q) & (q -> p)))
(((- P) / Q) & ((- Q) / P))
```

Transformación a forma normal negativa

- Interiorización de negaciones:
  - Implementación: **`(interioriza-negaciones` F`)`**
  - Ejemplos:

```
> (interioriza-negacion '(- (- p)))
P
> (interioriza-negacion '(- (p & q)))
((- P) / (- Q))
> (interioriza-negacion '(- (p / q)))
((- P) & (- Q))
> (interioriza-negacion '(- (- (p / q))))
(P / Q)
> (interioriza-negacion '(- ((- p) / q)))
(P & (- Q))
```


Transformación a forma normal negativa

- Transformación a forma normal negativa:

```
(defun forma-normal-negativa (F)
  (interioriza-negacion
   (elimina-implicaciones
    (elimina-equivalencias F))))
```

- Ejemplos:

```
> (forma-normal-negativa '(p <-> q))
(((- P) / Q) & ((- Q) / P))
> (forma-normal-negativa '((p / (- q)) -> r))
(((- P) & Q) / R)
> (forma-normal-negativa '((p & (q -> r)) -> s))
(((- P) / (Q & (- R))) / S)
```

Forma normal conjuntiva

- Disyunciones extendidas:
  - Si F es un literal, entonces F es una disyunción extendida
  - Si F y G son disyunciones extendidas, entonces $(F \vee G)$ también lo es
  - Ejemplos:
    - $\neg p \vee (q \vee \neg r)$ es una disyunción extendida
    - $\neg p \vee (q \wedge \neg r)$ no es una disyunción extendida
- Fórmulas en forma normal conjuntiva:
  - Si F es una disyunción extendida, entonces F está en forma normal conjuntiva
  - Si F y G están en forma normal conjuntiva, entonces $(F \wedge G)$ también lo está
  - Ejemplos:
    - $(\neg p \vee q) \wedge (\neg q \vee p)$ está en forma normal conjuntiva
    - $(\neg p \vee q) \wedge (q \rightarrow p)$ no está en forma normal conjuntiva

Transformación en forma normal conjuntiva

- Objetivo: Dada una fórmula F, obtener una fórmula en forma normal conjuntiva G tal que $F \equiv G$
- Procedimiento $fnc(F)$
  - Transformación a forma normal negativa
  - Interiorización de las disyunciones

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$
$$(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r)$$

- Propiedades:
  - $fnc(F)$ es una forma normal conjuntiva
  - $fnc(F) \equiv F$

Transformación en forma normal conjuntiva

- Interiorización de las disyunciones:
  - Implementación: **`(interioriza-disyuncion F)`**
  - Ejemplos

```
> (interioriza-disyuncion '(p / (q & r)))
((P / Q) & (P / R))
> (interioriza-disyuncion '((p & q) / r))
((P / R) & (Q / R))
```

- Transformación a forma normal conjuntiva:

```
(defun forma-normal-conjuntiva (F)
  (interioriza-disyuncion (forma-normal-negativa F)))
```

- Ejemplos:

```
> (forma-normal-conjuntiva '(p & (q -> r)))
(P & ((- Q) / R))
> (forma-normal-conjuntiva '(- (p & (q -> r))))
(((- P) / Q) & ((- P) / (- R)))
```


Forma normal disyuntiva

- Conjunciones extendidas:
  - Si F es un literal, entonces F es una conjunción extendida
  - Si F y G son conjunciones extendidas, entonces $(F \wedge G)$ también lo es
  - Ejemplos:
    - $\neg p \wedge (q \wedge \neg r)$ es una conjunción extendida
    - $\neg p \vee (q \wedge \neg r)$ no es una conjunción extendida
- Fórmulas en forma normal disyuntiva:
  - Si F es una conjunción extendida, entonces F está en forma normal disyuntiva
  - Si F y G están en forma normal disyuntiva, entonces $(F \vee G)$ también lo está
  - Ejemplos:
    - $(\neg p \wedge q) \vee (\neg q \wedge p)$ está una forma normal disyuntiva
    - $(\neg p \wedge q) \vee (q \rightarrow p)$ no está una forma normal disyuntiva

Transformación en forma normal disyuntiva

- Objetivo: Dada una fórmula F, obtener una fórmula en forma normal disyuntiva G tal que $F \equiv G$.
- Procedimiento $fnd(F)$
  - Transformación a forma normal negativa
  - Interiorización de las conjunciones
    $$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$
    $$(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r)$$
- Propiedades:
  - $fnd(F)$ es una forma normal disyuntiva
  - $fnd(F) \equiv F$

Transformación en forma normal disyuntiva

- Interiorización de las conjunciones:
  - Implementación: **`(interioriza-conjuncion F)`**
  - Ejemplos
    ```
    > (interioriza-conjuncion '(p & (q / r)))
    ((P & Q) / (P & R))
    > (interioriza-conjuncion '((p / q) & r))
    ((P & R) / (Q & R))
    ```
- Transformación a forma normal disyuntiva:
  ```
  (defun forma-normal-disyuntiva (F)
    (interioriza-conjuncion (forma-normal-negativa F)))
  ```
- Ejemplos:
  ```
  > (forma-normal-disyuntiva '(p & (q -> r)))
  ((P & (- Q)) / (P & R))
  > (forma-normal-disyuntiva '(- (p & (q -> r))))
  ((- P) / (Q & (- R)))
  ```

Ejercicios

- *Sea $F = (p \leftrightarrow q) \rightarrow [(q \vee r) \vee (\neg p \wedge \neg s)]$. Se pide determinar fórmulas equivalentes a F en Forma Normal Negativa, Forma Normal Disyuntiva y Forma Normal Conjuntiva*
- *Sea $F = [(p \vee r) \wedge (q \rightarrow r)] \leftrightarrow (\neg p \vee q)$. Se pide determinar fórmulas equivalentes a F en Forma Normal Negativa, Forma Normal Disyuntiva y Forma Normal Conjuntiva*

Bibliografía

- Alonso Jiménez, J.A. *Lógica computacional* (Univ. de Sevilla, 1997)
  - Cap. 5: “Equivalencias y formas normales”
- Chang, C.L. y Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving*. (Academic Press, 1973)
  - Cap. 2: “The propositional logic”
- Fitting, M. *First–Order Logic and Automated Theorem Proving (2nd, ed.)* (Springer, 1996)
  - Cap. 2: “Propositional Logic”
- Genesereth, M.R. y Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. (Morgan Kaufmann, 1987)
  - Cap. 2: “Propositional Logic”
- Paulson, L. *Logic and Proof* (University of Cambridge, 2003) **`http://www.cl.cam.ac.uk/Teaching/2003/LogicProof`**
  - Cap. 2: “Propositional logic”

Capítulo 5

Tableros semánticos

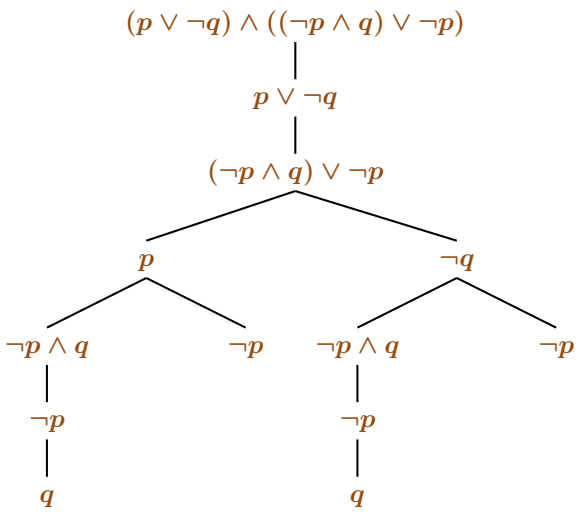
Lógica y Programación

Tableros Semánticos

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Descomposición de fórmulas



Notación uniforme

- Clasificación de fórmulas
  - Fórmulas con comportamiento conjuntivo: α
  - Fórmulas con comportamiento disyuntivo: β
  - Dobles negaciones: $\neg\neg G$
    - Implementación: **`(es-doble-negacion` F`)`**
    - Ejemplos:

```
> (es-doble-negacion '(- (- p)))
T
> (es-doble-negacion '(- ((- p) & (- q))))
NIL
```

  - Literales
    - Implementación: **`(es-literal` L`)`**

Notación uniforme

- Fórmulas con comportamiento conjuntivo: α

α	α_1	α_2
$(F \wedge G)$	F	G
$\neg(F \vee G)$	$\neg F$	$\neg G$
$\neg(F \rightarrow G)$	F	$\neg G$

$$\alpha \equiv \alpha_1 \wedge \alpha_2$$

- Implementación: **`(es-formula-alfa` F`)`**
- Ejemplos

```
> (es-formula-alfa '((p & ((- p) / q)) & (- q)))
T
> (es-formula-alfa '(p & ((- p) / q)))
T
> (es-formula-alfa '((- p) / q))
NIL
```

Notación uniforme

- Componentes de las fórmulas α
- Implementación: **(componentes-alfa α)**

```
> (componentes-alfa '((p & ((- p) / q)) & (- q)))
((P & ((- P) / Q)) (- Q))
> (componentes-alfa '(p & ((- p) / q)))
(P ((- P) / Q))
> (componentes-alfa '((- p) / q))
NIL
```


Notación uniforme

- Fórmulas con comportamiento disyuntivo: β

β	β_1	β_2
$(F \vee G)$	F	G
$\neg(F \wedge G)$	$\neg F$	$\neg G$
$(F \rightarrow G)$	$\neg F$	$\neg G$

$$\beta \equiv \beta_1 \vee \beta_2$$

- Implementación: **(es-formula-beta F)**
- Ejemplos

```
> (es-formula-beta '((p & ((- p) / q)) & (- q)))
NIL
> (es-formula-beta '(p & ((- p) / q)))
NIL
> (es-formula-beta '((- p) / q))
T
```

Notación uniforme

- Equivalencias: α o β

F	α_1	α_2	β_1	β_2
$(F \leftrightarrow G)$	$\neg F \vee G$	$F \vee G$	$F \wedge G$	$\neg F \wedge \neg G$
$\neg(F \leftrightarrow G)$	$F \vee G$	$\neg F \vee \neg G$	$\neg F \wedge G$	$F \wedge \neg G$

- Serán consideradas fórmulas β
- Ejemplos:

```
> (es-formula-beta '((p & q) <-> (p / (- r))))
T
> (es-formula-alfa '(- ((p / (- q)) <-> (- r))))
NIL
```

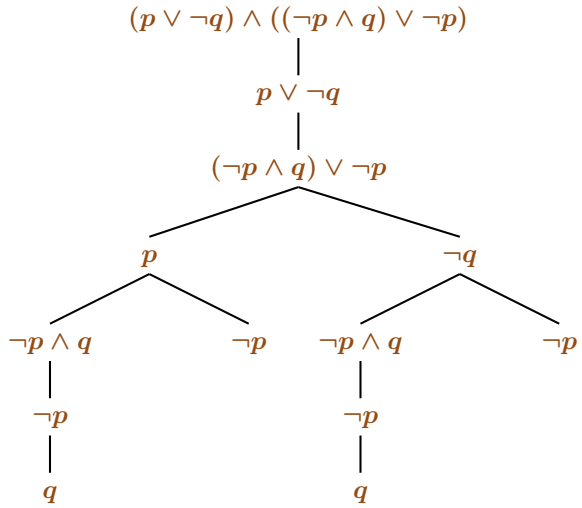

Notación uniforme

- Componentes de las fórmulas β
- Implementación: `(componentes-beta` β`)`

```
> (componentes-beta '((- p) / q))
((- P) Q)
> (componentes-beta '((p & q) <-> (- r)))
(((P & Q) & (- R)) ((- (P & Q)) & (- (- R))))
> (componentes-beta '(- (p / q)))
NIL
```

Tableros semánticos

- Estructura de árbol
- Reglas de expansión
- Ramas finales
- Ramas cerradas



Tableros semánticos

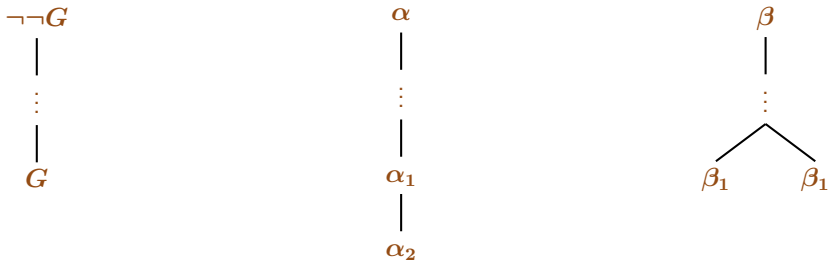
- Sea $\mathcal{S} = \{F_1, \ldots, F_n\}$ un conjunto finito de fórmulas.
  - El siguiente árbol de una rama es un tablero para $\mathcal{S}$:

$$F_1$$
$$F_2$$
$$\vdots$$
$$F_n$$

  - Si T es un tablero para $\mathcal{S}$ y T^* se obtiene a partir de T mediante una **regla de expansión** de tableros, entonces T^* también es un tablero para $\mathcal{S}$
- Semántica asociada
  - Rama: Conjunción de las fórmulas de los nodos
  - Árbol: Disyunción de las ramas

Tableros semánticos

- Reglas de expansión de tableros: dada una ocurrencia de una fórmula no literal F en una rama del tablero, dicha rama se expande con las componentes de F de la siguiente forma
  - Doble negación:
  - Fórmulas α:
  - Fórmulas β:



- Las reglas de expansión preservan la semántica de los tableros

Tableros semánticos

- Una rama de un tablero se dice **final** si todas las fórmulas no literales que aparecen en ella han sido expandidas
- Una rama de un tablero se dice **cerrada** si contiene fórmulas complementarias
  - Implementación: `(es-rama-cerrada` $\mathcal{S}$`)`
  - Ejemplos

```
> (es-rama-cerrada '(p (- q) (p & q)))
NIL
> (es-rama-cerrada '((- p) q (- (- p))))
T
```

- Un tablero se dice **final** si todas sus ramas son finales o cerradas
- Un tablero se dice **cerrado** si todas sus ramas son cerradas
- Propiedades semánticas
  - Una rama cerrada es un conjunto inconsistente de fórmulas
  - Un tablero cerrado es “inconsistente”

Tableros semánticos

- Un conjunto de fórmulas $\mathcal{S}$ es inconsistente si y sólo si $\mathcal{S}$ tiene un tablero cerrado
- Procedimiento de cálculo:
  - Considerar el tablero de una única rama con nodos etiquetados con las fórmulas de $\mathcal{S}$
  - Aplicar reglas de expansión de tableros hasta obtener un tablero final
  - Si el último tablero obtenido es cerrado entonces $\mathcal{S}$ es inconsistente

Tableros semánticos

- Implementación:
  - Las ramas del tablero se representan como listas de fórmulas
  - Las reglas de expansión de tableros seleccionan una fórmula no literal de una rama y la sustituyen por sus componentes
  - Se eliminan las repeticiones en las ramas
  - Se examinan todas las ramas hasta encontrar una final no cerrada o hasta que se terminan
  - Las ramas pendientes de examinar se encuentran en llamadas recursivas pendientes

Tableros semánticos

- Ejemplo (traza)

```
> (inconsistente-tableros '( ((p & ((- p) / (- q))) & (- (- q))) ))
    (inconsistente-tableros '( (p & ((- p) / (- q))) (- (- q)) ))
    |  (inconsistente-tableros '( p ((- p) / (- q)) (- (- q)) ))
    |  |  (inconsistente-tableros '( p (- p) (- (- q)) ))
    |  |  T
    |  |  (inconsistente-tableros '( p (- q) (- (- q)) ))
    |  |  |  (inconsistente-tableros '( p (- q) q ))
    |  |  |  T
    |  |  T
    |  T
    T
T
```


Tableros semánticos

```
(defun inconsistente-tableros (S)
  (let (F)
    (cond ((es-rama-cerrada S) T)
          ((setf F (find-if #'es-doble-negacion S))
           (inconsistente-tableros
            (adjoin (componente-doble-negacion F)
                    (remove F S :test #'equal) :test #'equal)))
          ((setf F (find-if #'es-formula-alfa S))
           (inconsistente-tableros
            (union (componentes-alfa F)
                   (remove F S :test #'equal) :test #'equal)))
          ((setf F (find-if #'es-formula-beta S))
           (and (inconsistente-tableros
                 (adjoin (first (componentes-beta F))
                         (remove F S :test #'equal) :test #'equal))
                (inconsistente-tableros
                 (adjoin (second (componentes-beta F))
                         (remove F S :test #'equal) :test #'equal))))
          (t nil))))
```

Obtención de modelos

- Toda rama final no cerrada proporciona un modelo: La interpretación formada por los literales de dicha rama
- Implementación: **(modelo-por-tableros $\mathcal{S}$)**
- Ejemplos

```
> (modelo-tableros '( (- ((p & (p -> q)) -> q)) ))
NIL
> (modelo-tableros '( (p & q) ))
(P Q)
> (modelo-tableros '( ((p -> q) & (q -> p)) ))
((- Q) (- P))
```


Forma normal disyuntiva

- Toda rama final no cerrada da lugar a una conjunción extendida
- Un tablero final no cerrado da lugar a una forma normal disyuntiva
- Implementación: **(fnd-tableros F)**
- Ejemplos

```
> (fnd-tableros '( (- ((p & (p -> q)) -> q)) ))
NIL
> (fnd-tableros '( (p & q) ))
(P Q)
> (fnd-tableros '( ((p -> q) & (q -> p)) ))
(((- Q) & (- P)) / (P & Q))
```

Tableros semánticos: Variantes

- Mejora en el tratamiento de las equivalencias

F	β_1		β_2	
	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{2,1}$	$\alpha_{2,2}$
$(F \leftrightarrow G)$	F	G	$\neg F$	$\neg G$
$\neg(F \leftrightarrow G)$	$\neg F$	G	F	$\neg G$

- Simplificación de fórmulas por literales
- Tratamiento de los nodos duplicados

Simplificación por literales

- Simplificar una fórmula asumiendo que un literal es cierto: $simp(F, L)$
  - $simp(L, L) = 1$
  - $simp(\neg L, L) = 0$
  - $$simp(\neg F, L) = \begin{cases} 1 & \text{si } simp(F, L) = 0 \\ 0 & \text{si } simp(F, L) = 1 \\ \neg simp(F, L) & \text{en otro caso} \end{cases}$$
  - $$simp(F_1 \wedge F_2, L) = \begin{cases} simp(F_2, L) & \text{si } simp(F_1, L) = 1 \\ 0 & \text{si } simp(F_1, L) = 0 \\ simp(F_1, L) & \text{si } simp(F_2, L) = 1 \\ 0 & \text{si } simp(F_2, L) = 0 \\ simp(F_1, L) \wedge simp(F_2, L) & \text{en otro caso} \end{cases}$$

Simplificación por literales

- Simplificar una fórmula asumiendo que un literal es cierto: $simp(F, L)$
  - $simp(F_1 \vee F_2, L) =$

$$\begin{cases} 1 & \text{si } simp(F_1, L) = 1 \\ simp(F_2, L) & \text{si } simp(F_1, L) = 0 \\ 1 & \text{si } simp(F_2, L) = 1 \\ simp(F_1, L) & \text{si } simp(F_2, L) = 0 \\ simp(F_1, L) \vee simp(F_2, L) & \text{en otro caso} \end{cases}$$

  - $simp(F_1 \rightarrow F_2, L) =$

$$\begin{cases} simp(F_2, L) & \text{si } simp(F_1, L) = 1 \\ 1 & \text{si } simp(F_1, L) = 0 \\ 1 & \text{si } simp(F_2, L) = 1 \\ \neg simp(F_1, L) & \text{si } simp(F_2, L) = 0 \\ simp(F_1, L) \rightarrow simp(F_2, L) & \text{en otro caso} \end{cases}$$

Simplificación por literales

- Simplificar una fórmula asumiendo que un literal es cierto: $simp(F, L)$
  - $simp(F_1 \leftrightarrow F_2, L) =$

$$\begin{cases} simp(F_2, L) & \text{si } simp(F_1, L) = 1 \\ simp(\neg F_2, L) & \text{si } simp(F_1, L) = 0 \\ simp(F_1, L) & \text{si } simp(F_2, L) = 1 \\ \neg simp(F_1, L) & \text{si } simp(F_2, L) = 0 \\ simp(F_1, L) \leftrightarrow simp(F_2, L) & \text{en otro caso} \end{cases}$$

Ejercicios

- *Sean $\mathcal{S} = \{p \to (q \to \neg r), p \vee s, (\neg q \wedge r) \to s\}$ y $F = s \vee \neg r$. Se pide comprobar que $\mathcal{S} \models F$ mediante tableros semánticos*
- *Sean $\mathcal{S} = \{(p \wedge \neg q) \to r, (\neg p \vee r) \to s, q \to \neg r\}$ y $F = r \to s$. Se pide comprobar que $\mathcal{S} \models F$ mediante tableros semánticos*
- *Sean $\mathcal{S} = \{a \to \neg b, \neg a \to (c \vee b), b \to (d \wedge a), d \to \neg c\}$ y $F = a \vee \neg d$. Se pide comprobar que $\mathcal{S} \models F$ mediante tableros semánticos*
- *Sean $\mathcal{S} = \{a \to \neg b, \neg b \to (c \vee d), c \to \neg d, d \to (b \wedge \neg a)\}$ y $F = b \vee \neg d$. Se pide comprobar que $\mathcal{S} \models F$ mediante tableros semánticos*

Bibliografía

- Ben–Hari, M. *Mathematical Logic for Computer Science (2nd, ed.)* (Springer, 2001)
  - Cap. 2: “Propositional calculus: formulas, models, tableaux”
- Fitting, M. *First–Order Logic and Automated Theorem Proving (2nd, ed.)* (Springer, 1996)
  - Cap. 2: “Propositional Logic”

Capítulo 6

Diagramas de decisión binarios

Lógica y Programación

Diagramas de Decisión Binarios

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Forma normal condicional

- Definición:

$$x \rightarrowtail y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

- Todas las conectivas lógicas se pueden expresar en función de $\rightarrowtail$ y los valores de verdad 0 y 1, de forma que las *condiciones* se evalúan únicamente sobre variables:
  - $x \equiv x \rightarrowtail 1, 0$
  - $\neg x \equiv x \rightarrowtail 0, 1$
  - $x \wedge y \equiv x \rightarrowtail (y \rightarrowtail 1, 0), 0$
  - $x \vee y \equiv x \rightarrowtail 1, (y \rightarrowtail 1, 0)$
  - $x \rightarrow y \equiv x \rightarrowtail (y \rightarrowtail 1, 0), 1$
  - $x \leftrightarrow y \equiv x \rightarrowtail (y \rightarrowtail 1, 0), (y \rightarrowtail 0, 1)$
- Una Forma Normal Condicional es una fórmula construida enteramente con la conectiva $\rightarrowtail$ y los valores de verdad 0 y 1, de forma que las *condiciones* se evalúan únicamente sobre variables

Expansión de Shannon

- $F[x/v]$ representa la fórmula obtenida a partir de F sustituyendo x por v, donde $v \in \{0, 1\}$
- Expansión de Shannon de F con respecto a x:

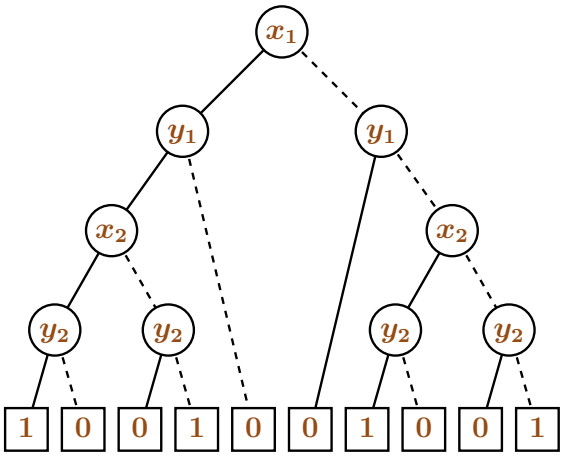
$$F \equiv x \rightarrowtail F[x/1], F[x/0]$$

- Transformación a forma normal condicional usando la expansión de Shannon
- Implementación: **`(expansion-shannon F)`**

Árboles de decisión

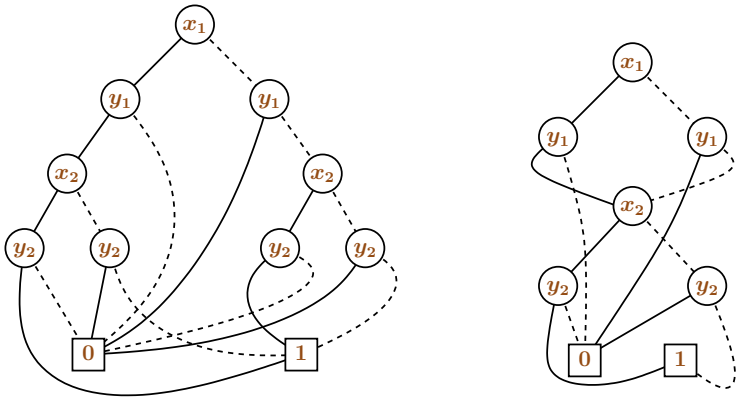
- Ejemplo: $F = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$

- $F = x_1 \rightarrowtail F_1, F_0$
- $F_1 = y_1 \rightarrowtail F_{11}, 0$
- $F_0 = y_1 \rightarrowtail 0, F_{00}$
- $F_{11} = x_2 \rightarrowtail F_{111}, F_{110}$
- $F_{00} = x_2 \rightarrowtail F_{001}, F_{000}$
- $F_{001} = y_2 \rightarrowtail 1, 0$
- $F_{000} = y_2 \rightarrowtail 0, 1$
- $F_{111} = y_2 \rightarrowtail 1, 0$
- $F_{110} = y_2 \rightarrowtail 0, 1$



Diagramas de decisión binarios

- Un diagrama de decisión binario (DDB) es un grafo acíclico dirigido con
  - Uno o dos nodos terminales etiquetados con $\mathbf{0}$ o $\mathbf{1}$
  - Un conjunto de nodos variables con dos hijos. Cada nodo variable está etiquetado con un símbolo proposicional.
- Ejemplo: $F = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$

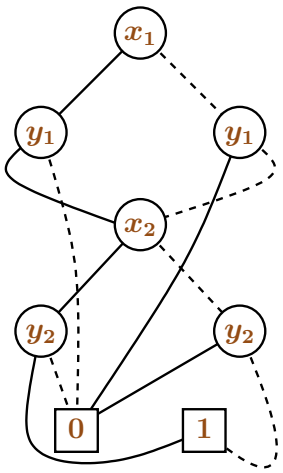


Reducción de Diagramas de Decisión Binarios

- Un DDB está reducido (DDBR) si
  - **unicidad**: No existen dos nodos en el diagrama etiquetados con el mismo símbolo proposicional y con los mismos hijos izquierdo y derecho (nodos duplicados)
  - **no-redundancia**: No existen nodos variables con hijos izquierdo y derecho idénticos (nodos redundantes)

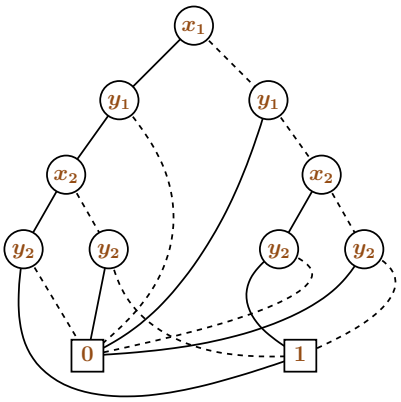
Reducción de Diagramas de Decisión Binarios

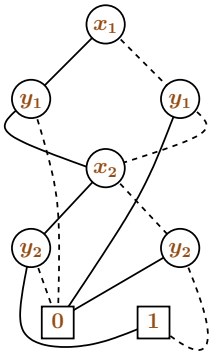
- Ejemplo: $F = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$



Diagramas de Decisión Binarios Ordenados

- Un DDB está ordenado (DDBO) si existe un orden entre los símbolos proposicionales $x_1 < x_2 < \ldots < x_n$ tal que para todo nodo variable etiquetado con x_i, si uno de sus nodos hijos es un nodo variable etiquetado con x_j entonces $x_i < x_j$
- Ejemplo: $F = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$
  $x_1 < y_1 < x_2 < y_2$





DDB ordenados y reducidos

- Diagramas de decisión binario reducido y ordenado (DDBRO)
- Propiedades: Una vez fijado el orden entre las variables
  - Toda fórmula es equivalente a un único DDBRO
  - Una fórmula es válida si su DDBRO equivalente está formado por el nodo terminal 1
  - Una fórmula es insatisfacible si su DDBRO equivalente está formado por el nodo terminal 0
  - Los modelos de una fórmula se pueden determinar a partir de los caminos en su DDBRO equivalente que van desde el nodo raíz hasta el nodo terminal 1

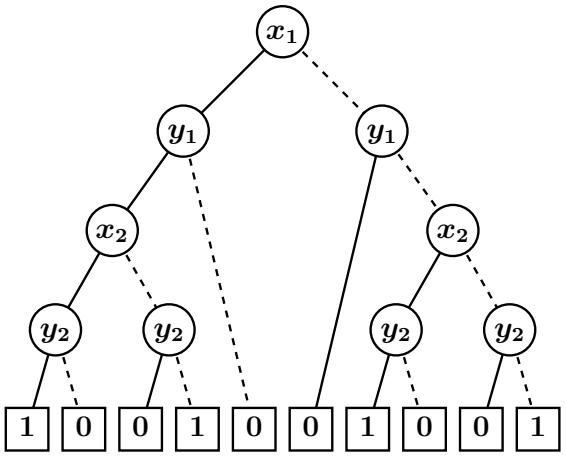
Representación de DDB

- Un DDB se representa con una lista de nodos, donde el primer elemento es el nodo raíz
- Los nodos terminales se representan con las listas `(1 1)` y `(0 0)`
- Un nodo variable se representa con una lista de cuatro elementos `(id,v,h1,h0)`,
  - `id` es un identificador único por nodo (números del 2 en adelante)
  - `v` es el símbolo proposicional que etiqueta el nodo
  - `h1` es el identificador del nodo correspondiente al valor 1 de `v`
  - `h0` es el identificador del nodo correspondiente al valor 0 de `v`

Representación de DDB

- Ejemplo: $F = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$

```
((2 x1 3 4)
 (3 y1 5 0)
 (4 y1 0 6)
 (5 x2 7 8)
 (6 x2 9 10)
 (7 y2 1 0)
 (8 y2 0 1)
 (9 y2 1 0)
 (10 y2 0 1)
 (1 1)
 (0 0))
```



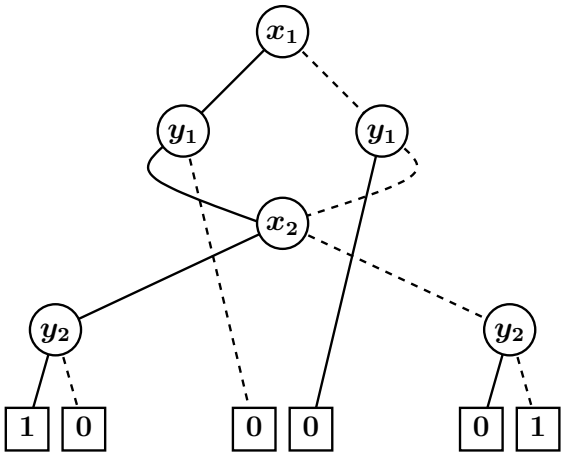
Reducción de DDB

- Ejemplo: $F = (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$

```
((2 x1 3 4)
 (3 y1 5 0)
 (4 y1 0 5)
 (5 x2 7 8)

 (7 y2 1 0)
 (8 y2 0 1)


 (1 1)
 (0 0))
```



Reducción de DDB

- Detección de nodos duplicados
  - Implementación: **(nodos-duplicados D)**
  - Ejemplos:

```
> (nodos-duplicados '((2 x1 0 1) (3 x1 0 1) (1 1) (0 0)))
((2 X1 0 1) (3 X1 0 1))
> (nodos-duplicados '((2 x1 0 1) (3 x2 0 1) (1 1) (1 1)))
((1 1) (1 1))
```

- Eliminación de nodos duplicados
  - Implementación: **(elimina-duplicados D)**
  - Ejemplos

```
> (elimina-duplicados '((2 x1 0 1) (3 x1 0 1) (1 1) (1 1)))
((2 X1 0 1) (1 1))
```


Reducción de DDB

- Detección de nodos redundantes
  - Implementación: **(nodo-redundante D)**
  - Ejemplos:

```
> (nodo-redundante '((2 x1 0 1) (3 x1 0 0) (1 1) (0 0)))
(3 X1 0 0)
> (nodo-redundante '((2 x1 3 3) (3 x2 0 1) (1 1) (1 1)))
(2 X1 3 3)
```

- Eliminación de nodos redundantes
  - Implementación: **(elimina-redundantes D)**
  - Ejemplos

```
> (elimina-redundantes '((2 x1 3 3) (3 x1 1 1) (1 1)))
((1 1))
```

Reducción de DDB

- La eliminación de nodos redundantes puede generar nodos duplicados
- La eliminación de nodos duplicados puede generar nodos redundantes
- Implementación: **`(reduce-ddb`** D**`)`**
- Ejemplos:

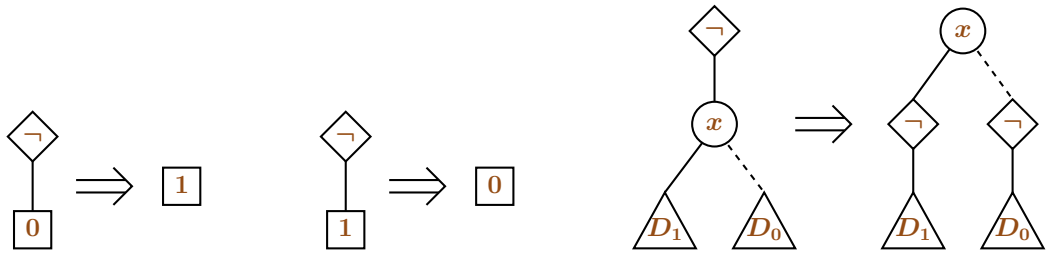
```
> (reduce-ddb '((2 x1 3 4) (3 y1 5 0) (4 y1 0 6) (5 x2 7 8)
                (6 x2 9 10) (7 y2 1 0) (8 y2 0 1) (9 y2 1 0)
                (10 y2 0 1) (1 1) (0 0))
((2 X1 3 4) (3 Y1 5 0) (4 Y1 0 5) (5 X2 7 8)
 (7 Y2 1 0) (8 Y2 0 1) (1 1) (0 0))
```


Operaciones con DDBRO

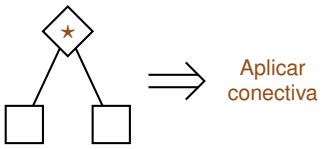
- Conmutatividad de $\twoheadrightarrow$ con respecto al resto de conectivas:
  - $\neg(x \twoheadrightarrow F_1, F_2) \equiv x \twoheadrightarrow \neg F_1, \neg F_2$
  - para $\star \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$
    - $(x \twoheadrightarrow F_1, F_2) \star G \equiv x \twoheadrightarrow F_1 \star G, F_2 \star G$
    - $(x \twoheadrightarrow F_1, F_2) \star (x \twoheadrightarrow G_1, G_2) \equiv x \twoheadrightarrow F_1 \star G_1, F_2 \star G_2$
- Construcción de un DDBRO para una fórmula F
  - Construir un árbol de decisión a partir de la tabla de verdad de F y reducirlo
  - Construir recursivamente DDBRO para las subfórmulas principales de F y aplicarles la conectiva principal de F

Operaciones con DDBRO

- Negación:

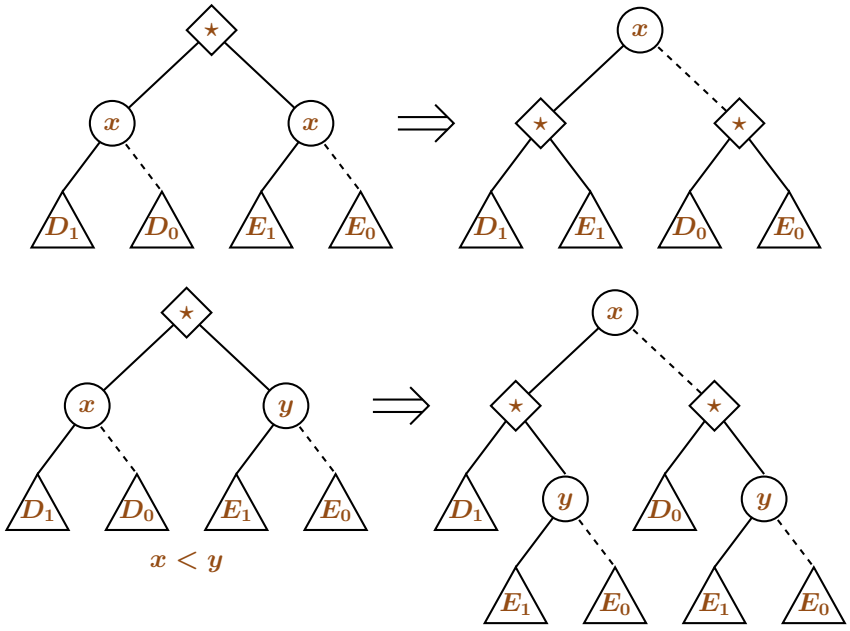


- Operadores binarios:



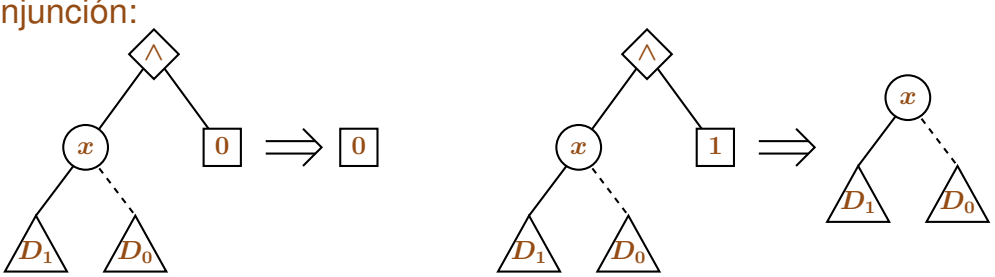
Operaciones con DDBRO

- Operadores binarios:

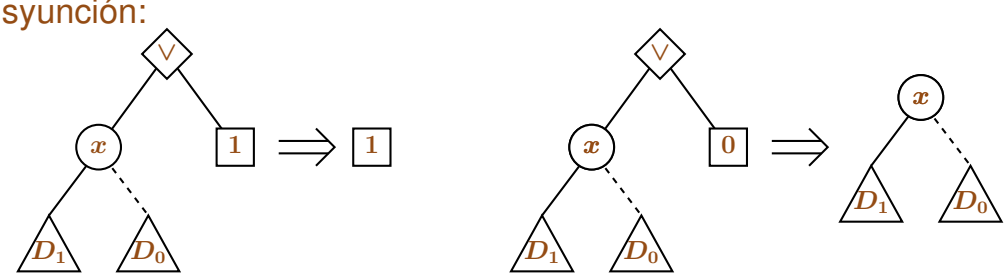


Operaciones con DDBRO

- Conjunción:

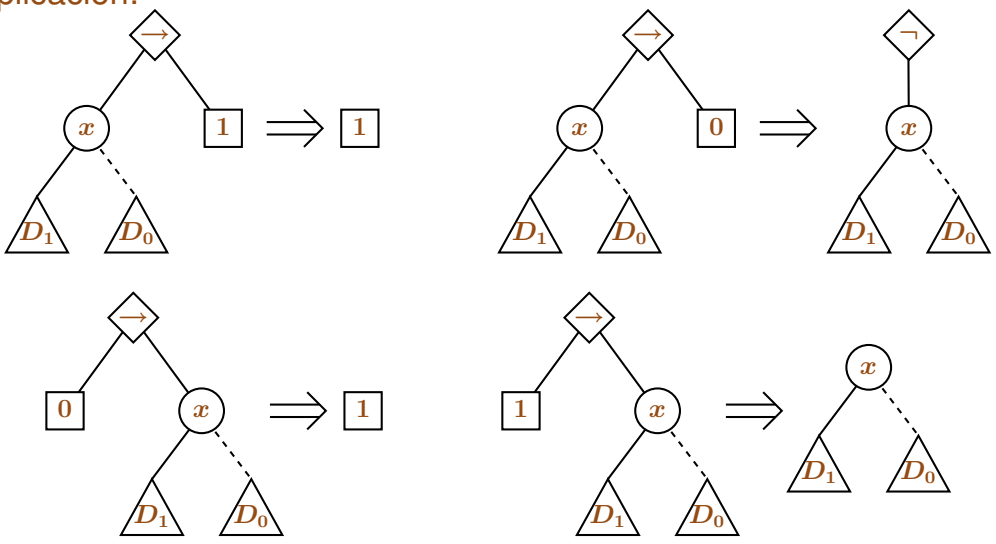


- Disyunción:



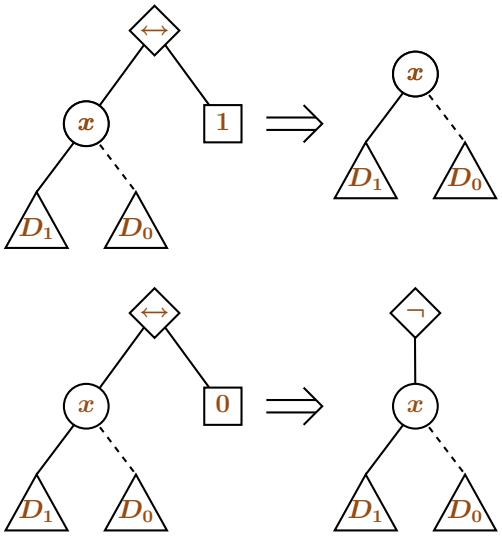
Operaciones con DDBRO

- Implicación:



Operaciones con DDBRO

- Equivalencia:



Operaciones con DDBRO

- Implementación:

```
(aplica-negacion D)
(aplica-conjuncion D E)
(aplica-disyuncion D E)
(aplica-implicacion D E)
(aplica-equivalencia D E)
```

Ejercicios

- *Consideremos las fórmulas $F_1 = (x_1 \wedge x_2) \rightarrow \neg x_3$ y $F_2 = \neg x_1 \leftrightarrow (x_2 \vee \neg x_4)$. Se pide:*
  1. *Construir diagramas de decisición binarios reducidos y ordenados para F_1 y F_2 con el siguiente orden entre las variables: $x_1 < x_2 < x_3 < x_4$*
  2. *Aplicar la operación conjunción a los diagramas anteriores*
- *Consideremos las fórmulas $F_1 = (x_1 \rightarrow x_2) \wedge \neg x_3$ y $F_2 = (\neg x_1 \vee x_4) \leftrightarrow \neg x_2$. Se pide:*
  1. *Construir diagramas de decisición binarios reducidos y ordenados para F_1 y F_2 con el siguiente orden entre las variables: $x_1 < x_2 < x_3 < x_4$*
  2. *Aplicar la operación implicación a los diagramas anteriores*

Bibliografía

- Bryant, R.E. Graph-based algorithms for Boolean function manipulation *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.
- Bryant, R.E. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

Capítulo 7

Cláusulas y formas clausales

Lógica y Programación

Cláusulas y formas clausales

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Cláusulas y formas clausales

- Cláusulas
  - Una cláusula es un conjunto de literales
  - Una cláusula se interpreta como la disyunción de todos sus literales
  - Ejemplos:

    $\{p, \neg q, r, \neg s\} \equiv p \vee \neg q \vee r \vee \neg s$
    $\{p, q, r\} \equiv p \vee q \vee r$
- Formas clausales
  - Una forma clausal es un conjunto de cláusulas
  - Una forma clausal se interpreta como la conjunción de todas sus cláusulas
  - Ejemplo:

    $\{\{p, q\}, \{r, \neg s\}\} \equiv (p \vee q) \wedge (r \vee \neg s)$

Transformación a cláusulas

- Cláusula de una disyunción extendida F:

$$clausula(F) = \begin{cases} \{F\}, & \text{si } F \text{ es un literal;} \\ clausula(F_1) \cup clausula(F_2), & \\ & \text{si } F = (F_1 \vee F_2) \end{cases}$$

- Implementación: **`(clausula F)`**
- Ejemplos:

```
> (clausula 'p)
(P)
> (clausula '(- p))
((- P))
> (clausula '(((- p) / r) / ((- p) / q)))
((- P) R Q)
```


Transformación a cláusulas

- Cláusulas de una fórmula en forma normal conjuntiva F:

$$clausulasFnc(F) = \begin{cases} clausulasFnc(F_1) \cup clausulasFnc(F_2), & \\ & \text{si } F = (F_1 \wedge F_2); \\ \{clausula(F)\}, & \text{en caso contrario} \end{cases}$$

- Implementación: **`(clausulas-fnc F)`**
- Ejemplos:

```
> (clausulas-fnc '(p & ((- q) / r)))
((P) ((- Q) R))
> (clausulas-fnc '(((- p) / q) & ((- p) / (- r))))
(((- P) Q) ((- P) (- R)))
```

Transformación a cláusulas

- Cláusulas de una fórmula F:
  $clausulas(F) = clausulasFnc(fnc(F))$
- Implementación:

```
(defun clausulas (F)
  (clausulas-fnc (forma-normal-conjuntiva F)))
```

- Ejemplos:

```
> (clausulas '(p & (q -> r)))
((P) ((- Q) R))
> (clausulas '(- (p & (q -> r))))
(((- P) Q) ((- P) (- R)))
```


Transformación a cláusulas

- Cláusulas de un conjunto de fórmulas $\mathcal{S}$:
  $clausulasConjunto(\mathcal{S}) = \bigcup\{clausulas(F) : F \in \mathcal{S}\}$
- Implementación: `(clausulas-conjunto` $\mathcal{S}$`)`
- Ejemplos:

```
> (clausulas-conjunto '((p -> q) (q -> r)))
(((- P) Q) ((- Q) R))
> (clausulas-conjunto '((p -> q) (q <-> p)))
(((- Q) P) ((- P) Q))
```

Símbolos proposicionales de cláusulas

- Símbolo proposicional de un literal:

$$sp(L) = \begin{cases} p & \text{si } L = p \\ p & \text{si } L = \neg p \end{cases}$$

- Implementación: **(simbolo-proposicional-literal L)**
- Ejemplos:

```
> (simbolo-proposicional-literal 'p)
P
> (simbolo-proposicional-literal '(- p))
P
```


Símbolos proposicionales de cláusulas

- Símbolos proposicionales de una cláusula:
  - Definición: $sp(C) = \{sp(L) : L \in C\}$
  - Implementación: **(simbolos-proposicionales-clausula C)**
  - Ejemplo:

```
> (simbolos-proposicionales-clausula '(p q (- p)))
(Q P)
```

- Símbolos proposicionales de un conjunto de cláusulas:
  - Definición: $sp(S) = \bigcup\{sp(C) : C \in S\}$
  - Implementación:
    (simbolos-proposicionales-clausulas S)
  - Ejemplo

```
> (simbolos-proposicionales-clausulas '((p q) ((- q) r)))
(P Q R)
```

Interpretaciones

- Interpretaciones de una cláusula:
  - $interpretaciones(C) = \{I : I \subseteq sp(C)\}$
  - Implementación: **(interpretaciones-clausula C)**
  - Ejemplos:

```
> (interpretaciones-clausula '(p q (- p)))
(NIL (P) (Q) (Q P))
> (interpretaciones-clausula '())
(NIL)
```

- Interpretaciones de un conjunto de cláusulas:
  - $interpretaciones(S) = \{I : I \subseteq sp(S)\}$
  - Implementación: **(interpretaciones-clausulas S)**
  - Ejemplos:

```
> (interpretaciones-clausulas '((p (- q)) ((- p) q)))
(NIL (Q) (P) (P Q))
> (interpretaciones-clausulas '())
(NIL)
```


Modelos de cláusulas

- Modelo de un literal: $I \models p \iff p \in I$
  $I \models \neg p \iff p \notin I$
- Implementación: **(es-modelo-literal I L)**
- Ejemplos

```
> (es-modelo-literal '(p r) 'p)
T
> (es-modelo-literal '(p r) 'q)
NIL
> (es-modelo-literal '(p r) '(- p))
NIL
> (es-modelo-literal '(p r) '(- q))
T
```

Modelos de cláusulas

- Modelo de una cláusula: $I \models C \Longleftrightarrow \exists L \in C, I \models L$
- Implementación: **(es-modelo-clausula** I C**)**
- Ejemplos:

```
> (es-modelo-clausula '(p r) '(p (- q)))
T
> (es-modelo-clausula '(r) '(p (- q)))
T
> (es-modelo-clausula '(q r) '(p (- q)))
NIL
> (es-modelo-clausula '(q r) '())
NIL
```

- Sea F una disyunción extendida, entonces $I \models F \Longleftrightarrow I \models clausula(F)$

Modelos de cláusulas

- Modelos de una cláusula: $modelos(C) = \{I \in interpretaciones(C) : I \models C\}$
- Implementación: **(modelos-clausula** C**)**
- Ejemplos:

```
> (modelos-clausula '((- p) q))
(NIL (Q) (P Q))
> (modelos-clausula '((- p) p))
(NIL (P))
> (modelos-clausula '())
NIL
```

- Sea F una disyunción extendida, entonces $modelos(F) = modelos(clausula(F))$

Modelos de cláusulas

- Modelo de un conjunto de cláusulas: $I \models S \Longleftrightarrow \forall C \in S, I \models C$
- Implementación: **(es-modelo-clausulas I S)**
- Ejemplos:

```
> (es-modelo-clausulas '(p r) '((p (- q)) (r)))
T
> (es-modelo-clausulas '(p) '((p (- q)) (r)))
NIL
> (es-modelo-clausulas '(p r) '())
T
```

- $I \models F \Longleftrightarrow I \models clausulas(F)$

Modelos de cláusulas

- Modelos de un conjunto de cláusulas:
  $modelos(S) = \{I \in interpretaciones(S) : I \models S\}$
- Implementación: **(modelos-clausulas S)**
- Ejemplos:

```
> (modelos-clausulas '(((- p) q) ((- q) p)))
(NIL (P Q))
> (modelos-clausulas '(((- p) q) (p) ((- q))))
NIL
> (modelos-clausulas '((p (- p) q)))
(NIL (Q) (P) (P Q))
```

- $modelos(F) = modelos(clausulas(F))$

Cláusulas válidas y satisfacibles

- Cláusulas válidas: $\models C \Longleftrightarrow \forall I \in interpretaciones(C), I \models C$
- $\models C$ si y sólo si $\exists L, \overline{L} \in C$
- Implementación: **`(es-clausula-valida C)`**
- Ejemplos:

```
> (es-clausula-valida '(p q (- p)))
T
> (es-clausula-valida '(p q (- r)))
NIL
> (es-clausula-valida '())
NIL
```

- La cláusula vacía $\emptyset$ no es válida
- $\models F \Longleftrightarrow \models clausulas(F)$

Cláusulas válidas y satisfacibles

- Cláusulas insatisfacibles:
  $\not\models C \Longleftrightarrow \nexists I \in interpretaciones(C), I \models C$
- $\not\models C$ si y sólo si $C = \emptyset$
- Implementación: **`(es-clausula-insatisfacible C)`**
- Ejemplos:

```
> (es-clausula-insatisfacible '(p q (- p)))
NIL
> (es-clausula-insatisfacible '(p q (- r)))
NIL
> (es-clausula-insatisfacible '())
T
```

Cláusulas válidas y satisfacibles

- Cláusulas satisfacibles: C es satisfacible
  $\Longleftrightarrow \exists I \in interpretaciones(C), I \models C$
- C es satisfacible si y sólo si $C \neq \emptyset$
- Implementación: **(es-clausula-satisfacible C)**
- Ejemplos:

```
> (es-clausula-satisfacible '(p q (- p)))
T
> (es-clausula-satisfacible '(p q (- r)))
T
> (es-clausula-satisfacible '())
NIL
```


Conjuntos de cláusulas válidos y consistentes

- Conjunto válido de cláusulas:
  $\models S \Longleftrightarrow \forall I \in interpretaciones(S), I \models S$
- $\models S \Longleftrightarrow \forall C \in S, \models C$
- Implementación: **(es-conjunto-clausulas-valido S)**
- Ejemplos:

```
> (es-conjunto-clausulas-valido '(((- p) q) ((- q) p)))
NIL
> (es-conjunto-clausulas-valido '(((- p) p) ((- q) q)))
T
> (es-conjunto-clausulas-valido '())
T
```

Conjuntos de cláusulas válidos y consistentes

- Conjunto consistente de cláusulas:
  S es consistente $\Longleftrightarrow \exists I \in interpretaciones(S), I \models S$
- Implementación: **`(es-conjunto-clausulas-consistente S)`**
- Ejemplos:

```
> (es-conjunto-clausulas-consistente '(((- p) q) ((- q) p)))
T
> (es-conjunto-clausulas-consistente '(((- p) q) (p) ((- q))))
NIL
```

- Sea $\mathcal{S}$ un conjunto de fórmulas, entonces $\mathcal{S}$ es consistente si y sólo si $clausulas(S)$ es consistente

Conjuntos de cláusulas válidos y consistentes

- Conjunto inconsistente de cláusulas:
  S es inconsistente $\Longleftrightarrow \nexists I \in interpretaciones(S), I \models S$
- Implementación: **`(es-conjunto-clausulas-inconsistente S)`**
- Ejemplos:

```
> (es-conjunto-clausulas-inconsistente '(((- p) q) ((- q) p)))
NIL
> (es-conjunto-clausulas-inconsistente '(((- p) q) (p) ((- q))))
T
```

- Sea $\mathcal{S}$ un conjunto de fórmulas, entonces $\mathcal{S}$ es inconsistente si y sólo si $clausulas(S)$ es inconsistente

Validez de fórmulas mediante cláusulas

- $\models F \Longleftrightarrow \models clausulas(F)$
- Implementación: **(es-valida-clausulas F)**
- Ejemplos:

```
> (es-valida-clausulas '(p -> p))
T
> (es-valida-clausulas '(p -> q))
NIL
> (es-valida-clausulas '((p -> q) / (q -> p)))
T
```


Consecuencia lógica mediante cláusulas

- Sean S_1 y S_2 dos conjuntos de cláusulas, entonces
  $S_1 \models S_2 \Longleftrightarrow$ los modelos de S_1 son modelos de S_2
- Implementación: **(es-consecuencia-entre-clausulas S_1 S_2)**
- Ejemplos:

```
> (es-consecuencia-entre-clausulas '(((- p) q) ((- q) r))
                                   '(((- p) r)))
T
> (es-consecuencia-entre-clausulas '((p)) '((p) (q)))
NIL
```

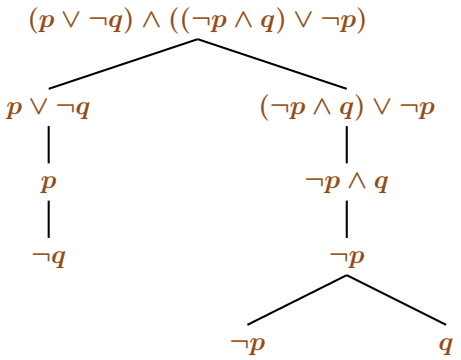
Consecuencia lógica mediante cláusulas

- Sea $\mathcal{S}$ un conjunto de fórmulas, entonces
  $\mathcal{S} \models F \iff clausulas(\mathcal{S}) \models clausulas(F)$
  $\iff clausulas(\mathcal{S} \cup \{\neg F\})$ es inconsistente
- Implementación: **`(es-consecuencia-clausulas` $\mathcal{S}$ F`)`**
- Ejemplos:

```
> (es-consecuencia-clausulas '((p -> q) (q -> r)) '(p -> r))
T
> (es-consecuencia-clausulas '(p) '(p & q))
NIL
```


Transformación a cláusulas (II)

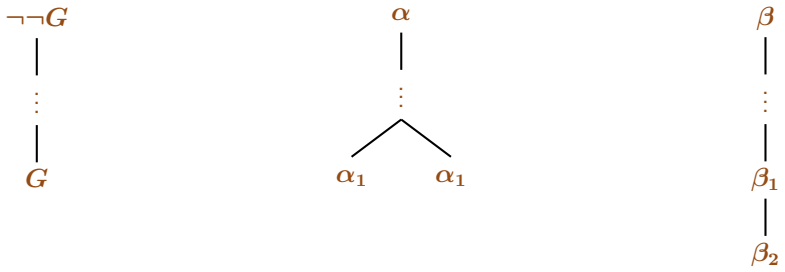
- Proceso similar al de los tableros semánticos
  - Estructura de árbol
  - Reglas de expansión
  - Ramas finales
  - Ramas cerradas



- Las ramas se interpretan como la disyunción de sus elementos
- El árbol se interpreta como la conjunción de sus ramas

Transformación a cláusulas (II)

- Reglas de expansión: dada una ocurrencia de una fórmula no literal F en una rama, dicha rama se expande con las componentes de F de la siguiente forma
  - Doble negación:
  - Fórmulas α:
  - Fórmulas β:



- El proceso de expansión preserva la interpretación de las ramas y del árbol

Transformación a cláusulas (II)

- Una rama se dice **final** si todas las fórmulas no literales que aparecen en ella han sido expandidas
- Una rama se dice **cerrada** si contiene fórmulas complementarias
  - Implementación: `(clausulas-uniforme F)`
  - Ejemplos

```
> (clausulas-uniforme '(p & ((- q) / r)))
((P) ((- Q) R))
> (clausulas-uniforme '(((- p) / q) & ((- p) / (- r))))
(((- P) Q) ((- P) (- R)))
> (clausulas-uniforme '((p / (- q)) & (((- p) & q) / (- p))))
((P (- Q)) ((- P)) (Q (- P)))
```

- Un árbol se dice **final** si todas sus ramas son finales o cerradas

Ejercicios

- *Sea* $F = (p \leftrightarrow q) \rightarrow [(q \vee r) \vee (\neg p \wedge \neg s)]$*. Se pide determinar un conjunto de cláusulas equivalente a* F
- *Sea* $F = [(p \vee r) \wedge (q \rightarrow r)] \leftrightarrow (\neg p \vee q)$*. Se pide determinar un conjunto de cláusulas equivalente a* F

Bibliografía

- Alonso Jiménez, J.A. *Lógica computacional* (Univ. de Sevilla, 1997)
  - Cap. 6: “Cláusulas proposicionales”
- Chang, C.L. y Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving*. (Academic Press, 1973)
  - Cap. 2: “The propositional logic”
- Fitting, M. *First–Order Logic and Automated Theorem Proving (2nd, ed.)* (Springer, 1996)
  - Cap. 2: “Propositional Logic”
- Genesereth, M.R. y Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. (Morgan Kaufmann, 1987)
  - Cap. 2: “Propositional Logic”
- Paulson, L. *Logic and Proof* (University of Cambridge, 2003) **`http://www.cl.cam.ac.uk/Teaching/2003/LogicProof`**
  - Cap. 2: “Propositional logic”

Capítulo 8

El procedimiento de Davis y Putnam

Lógica y Programación

El procedimiento de Davis y Putnam

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Simplificación por literales

- Simplificación de una cláusula por un literal:

$$C_L = \begin{cases} 1 & \text{si } L \in C \\ C - \{\overline{L}\} & \text{si } \overline{L} \in C \\ C & \text{en otro caso} \end{cases}$$

- Si $I \models L$ entonces $I \models C \iff I \models C_L$
- Implementación: **`(simplifica-clausula-literal C L)`**
- Ejemplos:

```
> (simplifica-clausula-literal '((- p) q r) 'q)
1
> (simplifica-clausula-literal '((- p) q r) '(- r))
((- P) Q)
> (simplifica-clausula-literal '((- p) q r) 's)
((- P) Q R)
```

Simplificación por literales

- Simplificación de un conjunto de cláusulas por un literal:

$$S_L = \{C - \{\overline{L}\} : C \in S, L \notin C\}$$

- Si $I \models L$ entonces $I \models S \Longleftrightarrow I \models S_L$
- Implementación: **`(simplifica-clausulas-literal S L)`**
- Ejemplos:

```
> (simplifica-clausulas-literal '((p q (- r)) (p (- q)) ((- p)))
                                'p)
(NIL)
> (simplifica-clausulas-literal '((p q (- r)) (p (- q)) ((- p)))
                                '(- q))
((P (- R)) ((- P)))
> (simplifica-clausulas-literal '((p q (- r)) (p (- q)) ((- p)))
                                'r)
((P Q) (P (- Q)) ((- P)))
```


Decisión de consistencia por bifurcación

- Bifurcación: Dado un conjunto de cláusulas S y un literal L se verifica S es consistente si y sólo si S_L o $S_{\overline{L}}$ es consistente
- Procedimiento de decisión:
  - Si $S = \emptyset$, entonces S es consistente
  - Si $\emptyset$ está en S, entonces S es inconsistente
  - Seleccionar un literal L que aparezca en S:
    - Evaluar recursivamente el proceso sobre S_L. Si S_L es consistente entonces S también lo es
    - Si S_L es inconsistente, evaluar recursivamente el proceso sobre $S_{\overline{L}}$. Si $S_{\overline{L}}$ es consistente entonces S también lo es
    - Si S_L y $S_{\overline{L}}$ son inconsistentes entonces S también lo es
- Heurística de selección de literal

Decisión de consistencia por bifurcación

- Implementación: **(inconsistente-bifuracion S)**
- Ejemplo (traza)

```
> (inconsistente-bifurcacion
                    '((p q (- r)) (p (- q)) ((- p)) (r) (u)))
    (inconsistente-bifurcacion '(nil (r) (u)))
    T
    (inconsistente-bifurcacion '((q (- r)) ((- q)) (r) (u)))
    |  (inconsistente-bifurcacion '(nil (r) (u)))
    |  T
    |  (inconsistente-bifurcacion '(((- r)) (r) (u)))
    |  |  (inconsistente-bifurcacion '(nil (u)))
    |  |  T
    |  |  (inconsistente-bifurcacion '(nil (u)))
    |  |  T
    |  T
    T
T
```


Extensión de interpretaciones

- Extensión de una interpretación I con un literal L

$$I_L = \begin{cases} I \cup \{p\} & \text{si } L = p \\ I - \{p\} & \text{si } L = \neg p \end{cases}$$

- $I_L \models L$
- Implementación: **(extiende-interpretacion I L)**
- Ejemplos:

```
> (extiende-interpretacion '(p q r) 's)
(S P Q R)
> (extiende-interpretacion '(p q r) 'q)
(P Q R)
> (extiende-interpretacion '(p q r) '(- s))
(P Q R)
> (extiende-interpretacion '(p q r) '(- q))
(P R)
```

Obtención de modelos por bifurcación

- Bifurcación: Dado un conjunto de cláusulas S, un literal L y una interpretación I se verifica $I \models S$ si y sólo si $I \models L$ e $I \models S_L$ o $I \models \overline{L}$ e $I \models S_{\overline{L}}$
- Procedimiento de obtención de modelos:
  - Si $S = \emptyset$, entonces $I = \emptyset$ es modelo de S
  - Si $\emptyset$ está en S, entonces S no tiene modelos
  - Seleccionar un literal L que aparezca en S:
    - Evaluar recursivamente el proceso sobre S_L. Si se obtiene un modelo I de S_L entonces I extendida con L es modelo de S
    - Si S_L no tiene modelos, evaluar recursivamente el proceso sobre $S_{\overline{L}}$. Si se obtiene un modelo I de $S_{\overline{L}}$ entonces I extendida con $\overline{L}$ es modelo de S
    - Si S_L y $S_{\overline{L}}$ no tienen modelos entonces S tampoco

Obtención de modelos por bifurcación

- Implementación: **`(modelo-bifuracion S)`**
- Ejemplos

```
> (modelo-bifurcacion '((p q)))
(P)
> (modelo-bifurcacion '((p q (- r)) (p (- q)) ((- p)) (r) (u)))
INCONSISTENTE
> (modelo-bifurcacion '((p q) ((- q)) ((- p) q (- r))))
(P)
> (modelo-bifurcacion '(((- p) q) (p) (r u)))
(P Q R)
> (modelo-bifurcacion '((p q) (p (- q)) (r q) (r (- q))))
(P R)
> (modelo-bifurcacion '((p q) (r (- s)) ((- r) s)))
(P R S)
```

Cláusulas tautológicas

- Una cláusula es tautológica si es válida $\Longleftrightarrow$ contiene literales complementarios
- Dado un conjunto de cláusulas S se tiene S es consistente si y sólo si $S - \{C : C \in S$ y C es tautológica$\}$ también lo es
- La eliminación de cláusulas tautológicas suele estar incorporada en el proceso de transformación a cláusulas $\Rightarrow$ disminuye el número de cláusulas generadas

Cláusulas unitarias

- Cláusulas unitarias: tienen un único literal
- Dada una interpretación I, $I \models \{L\}$ si y sólo si $I \models L$
- Dado un conjunto de cláusulas S, $C = \{L\} \in S$ e I una interpretación, entonces $I \models S$ si y sólo si $I \models L$ e $I \models S_L$
- Cláusulas unitarias en un conjunto de cláusulas S
  - Implementación: **`(busca-unitaria S)`**
  - Ejemplos:

```
> (busca-unitaria '((p q (- r)) (p (- q)) ((- p)) (r) (u)))
((- P))
> (busca-unitaria '((q (- r)) ((- q)) (r) (u)))
((- Q))
> (busca-unitaria '((r p) ((- r) u)))
NIL
```

Cláusulas unitarias

- Eliminación de cláusulas unitarias: Dado un conjunto de cláusulas S, S es consistente si y sólo si $S_{\{L,\{L\}\in S\}}$ también lo es
- Implementación: **`(elimina-clausulas-unitarias S)`**
- Ejemplos:

```
> (elimina-clausulas-unitarias
    '((p q (- r)) (p (- q)) ((- p)) (r) (u)))
(NIL)
> (elimina-clausulas-unitarias '((p q) ((- q)) ((- p) q (- r))))
NIL
> (elimina-clausulas-unitarias '(((- p) q) (p) (r u)))
((R U))
```

- Dado un conjunto de cláusulas S y una interpretación I, $I \models S$ si y sólo si I extendida con $\{L, \{L\} \in S\}$ es modelo de $S_{\{L,\{L\}\in S\}}$

Literales puros

- Dado un conjunto de cláusulas S, L es un literal puro en S si L aparece en S, pero $\overline{L}$ no
- Literales puros en un conjunto de cláusulas S
  - Implementación: **`(busca-literal-puro S)`**
  - Ejemplos:

```
> (busca-literal-puro '((p q) (p (- q)) (r q) (r (- q))))
P
> (busca-literal-puro '(((- p) q) (r (- s)) ((- r) s)))
(- P)
> (busca-literal-puro '(((- p) q) (p q) ((- q) p)))
NIL
```

Literales puros

- Eliminación de literales puros: Dado un conjunto de cláusulas S y L un literal puro en S, entonces S es consistente si y sólo si S_L es consistente
- Implementación: **`(elimina-literales-puros S)`**
- Ejemplos:

```
> (elimina-literales-puros '((p q) (p (- q)) (r q) (r (- q))))
NIL
> (elimina-literales-puros '((p q) (r (- s)) ((- r) s)))
((R (- S)) ((- R) S))
```

- Dado un conjunto de cláusulas S, L un literal puro en S y una interpretación I:
  - $I \models S$ entonces $I \models S_L$ ($\overline{L}$ no aparece en S)
  - $I \models S_L$ entonces I extendida con L es modelo de S

Factorización

- C_1 es un factor de C_2 si y sólo si $C_2 \subset C_1$
- Sea S un conjunto de cláusulas, $C_1, C_2 \in S$ tales que C_1 es un factor de C_2, entonces S es consistente si y sólo si $S - \{C_1\}$ también lo es
- Implementación: **`(elimina-factores S)`**
- Ejemplos:

```
> (elimina-factores '((p q) (p (- q)) (r p q) (s p (- q))))
((P Q) (P (- Q)))
> (elimina-factores '((p q) (s (- r)) ((- r) s)))
((P Q) (S (- R)) ((- R) S))
```

- Dado un conjunto de cláusulas S y una interpretación I, $I \models S$ si y sólo si I es modelo de $S - \{C : \exists C' \in S, C' \subset C\}$

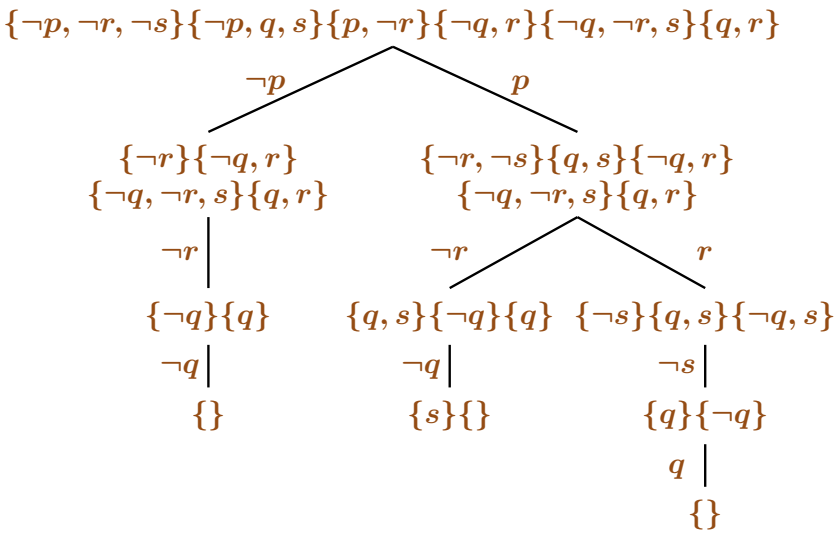
Cláusulas unitarias, literales puros y factores

- La eliminación de literales puros no genera nuevos factores ni cláusulas unitarias
- La eliminación de factores no genera nuevas cláusulas unitarias pero sí literales puros
- La eliminación de cláusulas unitarias puede generar nuevos literales puros y nuevos factores
- La bifurcación puede generar nuevos literales puros, cláusulas unitarias y factores

Procedimiento de Davis y Putnam

- Entrada: S, un conjunto de cláusulas no tautológicas
- Salida: T, si S es inconsistente; NIL, en caso contrario
- Procedimiento:
  - Eliminamos las cláusulas unitarias de S obteniendo S'
  - Eliminamos todos los factores de S' obteniendo S''
  - Eliminamos los literales puros de S'' obteniendo S^3
  - Si $S^3 = \emptyset$, entonces S^3 es consistente
  - Si $\emptyset$ está en S^3, entonces S^3 es inconsistente
  - Seleccionar un literal L que aparezca en S^3:
    - Evaluar recursivamente el proceso sobre S^3_L. Si S^3_L es consistente entonces S^3 también lo es
    - Si S^3_L es inconsistente, evaluar recursivamente el proceso sobre $S^3_{\overline{L}}$. Si $S^3_{\overline{L}}$ es consistente entonces S^3 también lo es
    - Si S^3_L y $S^3_{\overline{L}}$ son inconsistentes entonces S^3 también lo es

Procedimiento de Davis y Putnam



Procedimiento de Davis y Putnam

- Implementacion: **(inconsistente-dp S)**
- Ejemplos:

```
> (inconsistente-dp '((p q (- r)) (p (- q)) ((- p)) (r) (u)))
T
> (inconsistente-dp '((p q) ((- q)) ((- p) q (- r))))
NIL
> (inconsistente-dp '(((- p) q) (p) (r u)))
NIL
> (inconsistente-dp '((p q) (p (- q)) (r q) (r (- q))))
NIL
> (inconsistente-dp '((p q) (r (- s)) ((- r) s)))
NIL
> (inconsistente-dp '((p q) (p (- q)) (r q) (r (- q))))
NIL
```

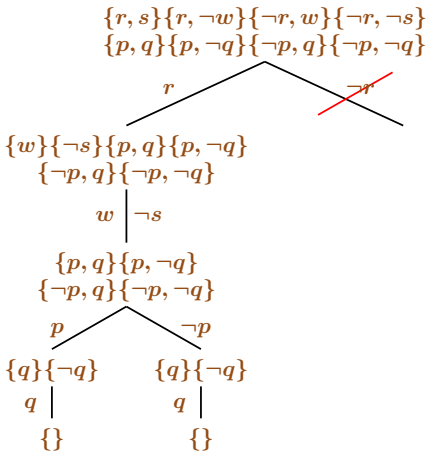
Procedimiento de Davis y Putnam

- Obtención de modelos: Los literales puros y los de cláusulas unitarias pasan a formar parte del modelo construido
- Implementación: **(modelo-dp S)**
- Ejemplos

```
> (modelo-dp '((p q) ((- q)) ((- p) q (- r))))
(P)
> (modelo-dp '((p (- q)) ((- p) q) (q (- r)) ((- q) (- r))))
(Q P)
> (modelo-dp '((p q (- r)) (p (- q)) ((- p)) (r) (u)))
INCONSISTENTE
> (modelo-dp '(((- p) q) (p) (r u)))
(R Q P)
> (modelo-dp '((p q) (p (- q)) (r q) (r (- q))))
(R P)
> (modelo-dp '((p q) (r (- s)) ((- r) s)))
(S R P)
```


Detección de bifurcaciones irrelevantes

- Dado S un conjunto de cláusulas y L un literal, la bifurcación de S en L es irrelevante si la inconsistencia de S_L no depende L
- Las bifurcaciones en literales puros son irrelevantes
- Ejemplo



Validez y consecuencia lógica

- Validez de fórmulas
  - Caracterización de validez mediante el método de Davis y Putnam $\models F \Longleftrightarrow \not\models clausulas(\neg F)$
  - Implementación: **(prueba-validez-dp F)**
- Consecuencia lógica
  - Caracterización de la consecuencia lógica mediante el método de Davis y Putnam $\mathcal{S} \models F \Longleftrightarrow \not\models clausulas(\mathcal{S} \cup \{\neg F\})$
  - Implementación: **(prueba-consecuencia-dp $\mathcal{S}$ F)**

Ejercicios

- *Consideremos el siguiente conjunto de cláusulas $S = \{p \vee q, q \vee r, r \vee w, \neg r \vee \neg p, \neg w \vee \neg q, \neg q \vee \neg r\}$. Se pide demostrar que es inconsistente mediante el procedimiento de Davis y Putnam*
- *Consideremos el siguiente conjunto de cláusulas $S = \{\neg p \vee \neg r \vee \neg s, \neg p \vee q \vee s, p \vee \neg r, \neg q \vee r, \neg q \vee \neg r \vee s, q \vee r\}$. Se pide demostrar que es inconsistente mediante el procedimiento de Davis y Putnam*

Bibliografía

- Davis, M. y Putnam, H. *A Computing Procedure for Quantification Theory*. (Journal of the ACM, 1960)
- Chang, C.L. y Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving*. (Academic Press, 1973)
  - Cap. 4.8.1: “The method of Davis and Putnam”

Capítulo 9

Resolución proposicional

Lógica y Programación

Resolución proposicional

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Introducción

- Idea básica de inconsistencia por resolución:
  - La cláusula vacía es insatisfacible
  - Un conjunto de cláusulas S es inconsistente syss la cláusula vacía es consecuencia de S
- Decisión de inconsistencia como proceso de búsqueda
  - Para determinar si S es inconsistente, generar consecuencias de S hasta que se obtenga la cláusula vacía

Regla de resolución proposicional

- Reglas de inferencia
  - Modus Ponens

$$\frac{p \rightarrow q \quad p}{q} \qquad \frac{\{\neg p, q\} \quad \{p\}}{\{q\}}$$

  - Modus Tollens

$$\frac{p \rightarrow q \quad \neg q}{\neg p} \qquad \frac{\{\neg p, q\} \quad \{\neg q\}}{\{\neg p\}}$$

  - Encadenamiento

$$\frac{p \rightarrow q \quad q \rightarrow r}{p \rightarrow r} \qquad \frac{\{\neg p, q\} \quad \{\neg q, r\}}{\{\neg p, r\}}$$

- Regla de resolución proposicional

$$\frac{\{p_1, \ldots, r, \ldots, p_m\} \quad \{q_1, \ldots, \neg r, \ldots, q_n\}}{\{p_1, \ldots, p_m, q_1, \ldots, q_n\}}$$

Regla de resolución proposicional

- Sean C_1 una cláusula, L un literal de C_1 y C_2 una cláusula que contiene el complementario de L. La resolvente de C_1 y C_2 respecto de L es $resolvente(C_1, C_2, L) = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$
- Implementación: **(resolvente C_1 C_2 L)**
- Ejemplos:

```
> (resolvente '((- p) q) '((- q) r) 'q)
((- P) R)
> (resolvente '((- p) (- q)) '(q r) '(- q))
((- P) R)
> (resolvente '((- p) q) '((- p) (- q)) 'q)
((- P))
```

Regla de resolución proposicional

- Resolventes de dos cláusulas: $resolventes(C_1, C_2)$ es el conjunto de las resolventes entre C_1 y C_2
- Si entre dos cláusulas hay más de una resolvente entonces todas las resolventes son cláusulas válidas
- Implementación: **(resolventes C_1 C_2)**
- Ejemplos:

```
> (resolventes '((- p) q) '(p (- q)))
NIL ; ((Q (- Q)) ((- P) P))
> (resolventes '((- p) q) '(p q))
((Q))
> (resolventes '((- p) q) '(q r))
NIL
```

- $\{\} \notin resolventes(\{p, q\}, \{\neg p, \neg q\})$

Regla de resolución proposicional

- Adecuación de la regla de resolución:
  $C \in resolventes(C_1, C_2) \Longrightarrow \{C_1, C_2\} \models C$
- Condición necesaria y suficiente de inconsistencia: Sean S un conjunto de cláusulas, $C_1, C_2 \in S$ y $C \in resolventes(C_1, C_2)$, entonces S es inconsistente si y sólo si $S \cup \{C\}$ es inconsistente
- Resolventes de una cláusula con un conjunto de cláusulas
  - Implementación: **(resolventes-clausula-conjunto C S)**
  - Ejemplo:

```
> (resolventes-clausula-conjunto '((- p) q)
                                 '((p q) (p r) ((- q) s)))
((Q) (Q R) ((- P) S))
```

Decisión de inconsistencia por resolución

- Entrada: S, un conjunto de cláusulas
- Salida: T, si S es inconsistente; NIL, en caso contrario
- Procedimiento:
  - Sea SOPORTE el conjunto S y USABLES el conjunto vacío (1)
  - Mientras SOPORTE sea no vacío, (2)
    - Sea ACTUAL la primera cláusula de SOPORTE (3)
    - Añadir ACTUAL al principio de USABLES (4)
    - Quitar ACTUAL de SOPORTE (5)
    - Sea NUEVAS las resolventes de ACTUAL con las cláusulas de USABLES (6)
    - Si una de las cláusulas de NUEVAS es la cláusula vacía, devolver T y terminar (7)
    - Añadir NUEVAS al final de SOPORTE (9)

Decisión de inconsistencia por resolución

- Implementación:

```
(defun inconsistente-resolucion-1 (S)
  (let ((soporte S)                                          ; 1
        (usables ())
        (actual ())
        (nuevas ()))
    (loop until (null soporte) do                            ; 2
      (setf actual (first soporte))                          ; 3
      (setf usables                                          ; 4
            (adjoin actual usables :test #'igual-conjunto))
      (setf soporte (rest soporte))                          ; 5
      (setf nuevas                                           ; 6
            (resolventes-clausula-conjunto actual usables))
      (when (member nil nuevas) (return t))                  ; 7
      (setf soporte                                          ; 8
            (union soporte nuevas :test #'igual-conjunto)))))
```

Decisión de inconsistencia por resolución

- Ejemplo (traza)

```
> (inconsistente-resolucion-1 '( ((- p) q) (p) ((- q)) ))
  | Soporte: (((- P) Q) (P) ((- Q)))                Usables: NIL
  | (resolventes-clausula-conjunto '((- p) q) '( ((- p) q) ))
  | NIL
  | Soporte: ((P) ((- Q)))                     Usables: (((- P) Q))
  | (resolventes-clausula-conjunto '(p) '( (p) ((- p) q) ))
  | ((Q))
  | Soporte: (((- Q)) (Q))                Usables: ((P) ((- P) Q))
  | (resolventes-clausula-conjunto '((- q))
  |                                '( ((- q)) (p) ((- p) q) ))
  | (((- P)))
  | Soporte: ((Q) ((- P)))          Usables: (((- Q)) (P) ((- P) Q))
  | (resolventes-clausula-conjunto '(q)
  |                                '( (q) ((- q)) (p) ((- p) q) ))
  | (NIL)
  T
```


Pruebas por resolución

- La sucesión $(C_1, \ldots, C_n)$ es una prueba por resolución de C a partir de S si $C = C_n$ y para todo $i \in \{1, ..., n\}$ se verifica una de las siguientes condiciones:
  - $C_i \in S$;
  - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$
- C es probable por resolución a partir de S si existe una prueba por resolución de C a partir de S
- Representación: $S \vdash C$
- Refutación por resolución:
  - La sucesión $(C_1, \ldots, C_n)$ es una refutación por resolución de S si es una prueba por resolución de la cláusula vacía a partir de S
  - S es refutable por resolución si existe una refutación por resolución a partir de S
  - Representación: $S \vdash \{\}$

Pruebas por resolución

- Refutación de $\{\{p, q\}, \{\neg p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}\}$:

```
1 NIL {P,Q}
2 NIL {-P,Q}
3 NIL {P,-Q}
4 NIL {-P,-Q}
5 (2 1) {Q}
7 (4 3) {-Q}
13 (7 5) {}
```

- Propiedades del cálculo por resolución:
  - Adecuación: $S \vdash \{\} \Longrightarrow S$ es inconsistente
  - Completitud: S es inconsistente $\Longrightarrow S \vdash \{\}$

Pruebas por resolución

- Búsqueda de refutaciones

```
> (prueba-1 '(((- p) q) (p) ((- q))))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {P}
  ** 4 (2 1) {Q}
3 NIL {-Q}
  ** 5 (3 1) {-P}
4 (2 1) {Q}
  ** 6 (4 3) {}
```

```
========== Prueba ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
4 (2 1) {Q}
6 (4 3) {}
===== Fin de la prueba =====
T
```

Cláusulas anotadas

- Estructura con tres campos: número, padres, cláusula
- Implementación:

```
(defstruct (clausula-anotada (:constructor crea-clausula-anotada)
                             (:conc-name ca-)
                        (:print-function escribe-clausula-anotada))
  (numero 0)
  padres
  clausula)
```

- Ejemplo

```
> (setf C (crea-clausula-anotada :numero 2 :clausula '((- p) q)))
2 NIL {-P,Q}
> (ca-numero x)
2
> (ca-padres x)
NIL
> (ca-clausula x)
((- P) Q)
```


Cláusulas anotadas

- Procedimiento de impresión en pantalla

```
(defun escribe-clausula-anotada (clausula-anotada
                                 &optional (canal t) profundidad)
  (format canal "~d ~s ~a"
          (ca-numero clausula-anotada)
          (ca-padres clausula-anotada)
          (clausula->cadena (ca-clausula clausula-anotada))))

(defun clausula->cadena (C)
  (if (null C)
      "{}"
      (format nil "{~a~{,~a~}}"
              (literal->cadena (first C))
              (mapcar #'literal->cadena (rest C)))))

(defun literal->cadena (L)
  (if (es-literal-positivo L)
      (format nil "~a" L)
      (format nil "-~a" (complementario L))))
```

Cláusulas anotadas

- Resolvente de cláusulas anotadas con respecto a un literal
  - Implementación: **(resolvente-ca Ca_1 Ca_2 L)**
  - Ejemplos:

```
> (setf ca1 (crea-clausula-anotada :numero 1 :clausula '((- p) q)))
1 NIL {-P,Q}
> (setf ca2 (crea-clausula-anotada :numero 2 :clausula '((- q) r)))
2 NIL {-Q,R}
> (resolvente-ca ca1 ca2 'q)
0 (1 2) {-P,R}
```

- Resolventes de cláusulas anotadas
  - Implementación: **(resolventes-ca Ca_1 Ca_2)**
  - Ejemplos:

```
> (resolventes-ca ca1 ca2)
(0 (1 2) {-P,R})
```


Cláusulas anotadas

- Resolventes de una cláusula anotada con un conjunto de cláusulas
  - Implementación: **(resolventes-ca-conjunto Ca Sa)**
  - Ejemplos:

```
> (setf ca1 (crea-clausula-anotada :numero 1 :clausula '((- p) q)))
1 NIL {-P,Q}
> (setf ca2 (crea-clausula-anotada :numero 2 :clausula '(p q)))
2 NIL {P,Q}
> (setf ca3 (crea-clausula-anotada :numero 3 :clausula '(p r)))
3 NIL {P,R}
> (setf ca4 (crea-clausula-anotada :numero 4 :clausula '((- q) s)))
4 NIL {-Q,S}
> (resolventes-ca-conjunto-1 ca1 (list ca2 ca3 ca4))
(0 (1 2) {Q} 0 (1 3) {Q,R} 0 (1 4) {-P,S})
```

Búsqueda de prueba por saturación

- Búsqueda de refutaciones

```
> (prueba-1 '(((- p) q) (p) ((- q))))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {P}
  ** 4 (2 1) {Q}
3 NIL {-Q}
  ** 5 (3 1) {-P}
4 (2 1) {Q}
  ** 6 (4 3) {}
```

```
========== Prueba ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
4 (2 1) {Q}
6 (4 3) {}
===== Fin de la prueba =====
T
```


Búsqueda de prueba por saturación

- Entrada: S, un conjunto de cláusulas
- Salida: Una refutación de S, si S es inconsistente; NIL, en otro caso
- Procedimiento:
  - Sea SOPORTE el conjunto S y USABLES el conjunto vacío (1)
  - Mientras SOPORTE sea no vacío y no se tenga una refutación, (2)
    - Sea ACTUAL la primera cláusula de SOPORTE (3)
    - Añadir ACTUAL al principio de USABLES (4)
    - Quitar ACTUAL de SOPORTE (5)
    - Sea NUEVAS las resolventes de ACTUAL con las cláusulas de USABLES (6)
    - Para cada cláusula C de NUEVAS (7)
      - Añadir C al final de SOPORTE (8)
      - Si C es la cláusula vacía, escribir la refutación y terminar (9)

Búsqueda de prueba por saturación

- Implementación:

```
(defun prueba-1 (S)
  (let ((soporte ()) (usables ()) (actual ()) (nuevas ()) (probada nil))
    (setf *contador* 0)
    (setf soporte (inicia-soporte S))                                    ; 1
    (loop until (null soporte) do                                        ; 2
      (when probada (return t))                                          ; 2
      (setf actual (first soporte))                                      ; 3
      (format t "~&~a" actual)
      (setf usables                                                      ; 4
            (adjoin actual usables :test #'igual-clausula-anotada))
      (setf soporte (rest soporte))                                      ; 5
      (setf nuevas (resolventes-ca-conjunto actual usables))             ; 6
      (loop for CA in nuevas do                                          ; 7
        (numera CA)
        (format t "~&  ** ~a" CA)
        (if (not (member CA soporte :test #'igual-clausula-anotada))
            (setf soporte (append soporte (list CA))))                   ; 8
        (when (null (ca-clausula CA))                                    ; 9
              (escribe-prueba CA usables)
              (return (setf probada t)))))))
```


Búsqueda de prueba por saturación

- Iniciación del soporte
  - Construir cláusulas anotadas e imprimirlas en pantalla
  - Implementación: **(inicia-soporte** Sa**)**
  - Variable global para enumerar las cláusulas: ***contador***
  - Ejemplo:

```
> (inicia-soporte '(((- p) q) (p) ((- q))))
========== Soporte ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
====== Fin del soporte ======

(7 NIL {-P,Q} 8 NIL {P} 9 NIL {-Q})
```

- Comparación de cláusulas anotadas
  - Implementación: **(igual-clausula-anotada** Ca_1 Ca_2**)**

Búsqueda de prueba por saturación

- Numeración de cláusulas anotadas
  - Implementación: **(numera** Ca**)**
- Escritura de la prueba
  - Determinar la secuencia ordenada de cláusulas que intervienen en la prueba e imprimirlas en pantalla
  - Implementación: **(escribe-prueba** Cv Cu**)**
  - Ejemplo:

```
> (escribe-prueba vacia usables)
========== Prueba ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
4 (2 1) {Q}
6 (4 3) {}
===== Fin de la prueba =====
T
```


Búsqueda de prueba mejorada

- Prueba por saturación

```
> (prueba-1 '((p) (q) (p q) ((- p) q) (p (- q)) ((- p) (- q))))

========== Soporte ==========
[1 NIL {P}]
[2 NIL {Q}]
[3 NIL {P,Q}]
[4 NIL {-P,Q}]
[5 NIL {P,-Q}]
[6 NIL {-P,-Q}]
====== Fin del soporte ======

[1 NIL {P}]
[2 NIL {Q}]
[3 NIL {P,Q}]
[4 NIL {-P,Q}]
  ** [7 (4 1) {Q}]
[5 NIL {P,-Q}]
  ** [8 (5 2) {P}]
[6 NIL {-P,-Q}]
  ** [9 (6 2) {-P}]
  ** [10 (6 1) {-Q}]
[7 (4 1) {Q}]
  ** [11 (7 6) {-P}]
  ** [12 (7 5) {P}]
[8 (5 2) {P}]
  ** [13 (8 6) {-Q}]
  ** [14 (8 4) {Q}]
[9 (6 2) {-P}]
  ** [15 (9 5) {-Q}]
  ** [16 (9 3) {Q}]
  ** [17 (9 1) {}]

========== Prueba ==========
[1 NIL {P}]
[2 NIL {Q}]
[6 NIL {-P,-Q}]
[9 (6 2) {-P}]
[17 (9 1) {}]
===== Fin de la prueba =====
T
```

Búsqueda de prueba mejorada

- Evitar la generación de cláusulas que ya están en el conjunto usables
- Analizar las cláusulas unitarias recién generadas por si dan lugar a la cláusula vacía
- Prueba mejorada

```
> (prueba-2 '((p) (q) (p q) ((- p) q) (p (- q)) ((- p) (- q))))

========== Soporte ==========
[1 NIL P]
[2 NIL Q]
[3 NIL P,Q]
[4 NIL -P,Q]
[5 NIL P,-Q]
[6 NIL -P,-Q]
====== Fin del soporte ======

[1 NIL P]
[2 NIL Q]
[3 NIL P,Q]
[4 NIL -P,Q]
[5 NIL P,-Q]
[6 NIL -P,-Q]
  ** [7 (6 2) -P]
[8 (7 1) ]

========== Prueba ==========
[1 NIL P]
[2 NIL Q]
[6 NIL -P,-Q]
[7 (6 2) -P]
[8 (7 1) ]
===== Fin de la prueba =====
T
```


Búsqueda de prueba mejorada

- Entrada: S, un conjunto de cláusulas
- Salida: Una refutación de S, si S es inconsistente; NIL, en otro caso
- Procedimiento:
  - Sea SOPORTE el conjunto S y USABLES el conjunto vacío
  - Mientras SOPORTE sea no vacío y no se tenga una refutación,
    - Sea ACTUAL la primera cláusula de SOPORTE
    - Añadir ACTUAL al principio de USABLES
    - Quitar ACTUAL de SOPORTE
    - Sea NUEVAS las resolventes de ACTUAL con las cláusulas de USABLES que no estén en USABLES ni en SOPORTE
    - Para cada cláusula C de NUEVAS
      - Añadir C al final de SOPORTE
      - Si C es la cláusula vacía, escribir la refutación y terminar
      - Si C es unitaria y su complementaria está en USABLES o en SOPORTE, escribir la refutación y terminar

Búsqueda de prueba mejorada

- Implementación:

```
(defun prueba-2 (S)
  (let ((soporte ()) (usables ()) (actual ()) (nuevas ()) (probada nil))
    (setf *contador* 0)
    (setf soporte (inicia-soporte S))
    (loop until (null soporte) do
      (when probada (return t))
      (setf actual (first soporte))
      (format t "~&~a" actual)
      (setf usables (adjoin actual usables :test #'igual-clausula-anotada))
      (setf soporte (rest soporte))
      (setf nuevas (nuevas-resolventes actual usables soporte))
      (loop for CA in nuevas do
        (numera CA)
        (format t "~&  ** ~a" CA)
        (setf soporte (append soporte (list CA)))
        (cond ((null (ca-clausula CA)) (escribe-prueba CA usables)
                                       (return (setf probada t)))
              ((and (es-unitaria (ca-clausula CA))
                    (complementaria CA usables soporte))
               (procesa-unitaria CA usables soporte)
               (return (setf probada t)))))))
```


Búsqueda de prueba mejorada

- Nuevas resolventes
  - Generar las resolventes entre una cláusula anotada C y un conjunto de cláusulas anotadas Cu, que no aparecen ya en Cu o en otro conjunto de cláusulas anotadas Cs
  - Implementación: **(nuevas-resolventes C Cu Cs)**
- Búsqueda de la cláusula complementaria
  - Dada una cláusula unitaria $C = \{L\}$, buscar su cláusula unitaria complementaria $\{\overline{L}\}$ en dos conjuntos de cláusulas anotadas
  - Implementación: **(complementaria C Cu Cs)**
- Prueba obtenida con una cláusula unitaria
  - Dada una cláusula unitaria $C = \{L\}$, cuya cláusula complementaria está en Cu o en Cs, escribir la refutación que se obtiene a partir de ella y las cláusulas de Cs
  - Implementación: **(procesa-unitaria C Cu Cs)**

Prueba de validez mediante resolución

- Reducción de validez a refutación por resolución:
  $\models F \Longleftrightarrow clausulas(\neg F) \vdash \{\}$
- Implementación: **(prueba-validez F)**
- Ejemplo:

```
> (prueba-validez '((p -> q) / (q -> p)))

========== Soporte ==========
1 NIL {P}
2 NIL {-Q}
3 NIL {Q}
4 NIL {-P}
====== Fin del soporte ======

1 NIL {P}
2 NIL {-Q}
3 NIL {Q}
  ** 5 (3 2) {}

========== Prueba ==========
2 NIL {-Q}
3 NIL {Q}
5 (3 2) {}
===== Fin de la prueba =====
T
```


Consecuencia mediante resolución

- Reducción de consecuencia a refutación por resolución:
  $\mathcal{S} \models F \Longleftrightarrow clausulas(\mathcal{S} \cup \{\neg F\}) \vdash \{\}$
- Implementación: **(prueba-consecuencia $\mathcal{S}$ F)**
- Ejemplo 1:

```
> (prueba-consecuencia '(p) '(p & q))

========== Soporte ==========
1 NIL {P}
2 NIL {-P,-Q}
====== Fin del soporte ======

1 NIL {P}
2 NIL {-P,-Q}
  ** 3 (2 1) {-Q}
3 (2 1) {-Q}
NIL
```

Consecuencia mediante resolución

- Ejemplo 2:

```
> (prueba-consecuencia '((p -> q) (q -> r)) '(p -> r))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {P}
4 NIL {-R}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {-Q,R}
  ** 5 (2 1) {R,-P}
3 NIL {P}
  ** 6 (3 1) {Q}
4 NIL {-R}
  ** 7 (4 2) {-Q}
8 (7 6) {}

========== Prueba ==========
1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {P}
4 NIL {-R}
6 (3 1) {Q}
7 (4 2) {-Q}
8 (7 6) {}
===== Fin de la prueba =====
T
```


Consecuencia mediante resolución

- Ejemplo 3:

```
> (prueba-consecuencia '((p  -> q) (m -> (p / q))) '(m -> q))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {-M,P,Q}
3 NIL {M}
4 NIL {-Q}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {-M,P,Q}
  ** 5 (2 1) {-M,Q}
3 NIL {M}
  ** 6 (3 2) {P,Q}
4 NIL {-Q}
  ** 7 (4 2) {-M,P}
  ** 8 (4 1) {-P}
5 (2 1) {-M,Q}
  ** 9 (5 3) {Q}
10 (9 4) {}

========== Prueba ==========
1 NIL {-P,Q}
2 NIL {-M,P,Q}
3 NIL {M}
4 NIL {-Q}
5 (2 1) {-M,Q}
9 (5 3) {Q}
10 (9 4) {}
===== Fin de la prueba =====
T
```

Ejercicios

- *Consideremos el siguiente conjunto de cláusulas* $S = \{p \vee q, q \vee r, r \vee w, \neg r \vee \neg p, \neg w \vee \neg q, \neg q \vee \neg r\}$. *Encontrar una refutación por resolución de* S
- *Consideremos el siguiente conjunto de cláusulas* $S = \{\neg p \vee \neg r \vee \neg s, \neg p \vee q \vee s, p \vee \neg r, \neg q \vee r, \neg q \vee \neg r \vee s, q \vee r\}$. *Encontrar una refutación por resolución de* S

Bibliografía

- Chang, C.L. y Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving*. (Academic Press, 1973)
  - Cap. 5: “The resolution principle”
- Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).
  - Cap. 2: “Logic and resolution”
- Rich, E. y Knight, K. *Inteligencia artificial (segunda edición)* (McGraw–Hill Interamericana, 1994)
  - Cap. 5.4.3: “Resolución en lógica proposicional”
- Russell, S. y Norvig, P. *Inteligencia artificial (un enfoque moderno)* (Prentice–Hall, 1996)
  - Cap. 9.5: “Resolución: un procedimiento completo de inferencia”

Capítulo 10

Refinamientos de resolución

Lógica y Programación

Refinamientos de resolución

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

P–resolución

- Cláusulas positivas
  - C es positiva si todos sus literales son positivos
  - Implementación: **`(es-ca-positiva` Ca`)`**
  - Ejemplos:

```
> (setf ca1 (crea-clausula-anotada :numero 1 :clausula '(p q r)))
1 NIL {P,Q,R}
> (es-ca-positiva ca1)
T
> (setf ca2 (crea-clausula-anotada :numero 1 :clausula '(p (- q))))
2 NIL {P,-Q}
> (es-ca-positiva ca2)
NIL
```

P–resolución

- Prueba por P–resolución
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por P–resolución de C a partir de S si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones:
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$ y alguna de las cláusulas C_j o C_k es positiva
  - C es probable por P–resolución a partir de S si existe una prueba por P–resolución de C a partir de S
  - Representación: $S \vdash_P C$
- Propiedades
  - $S \vdash_P \{\} \iff S$ es inconsistente

P–resolución

- Búsqueda de refutaciones

```
> (prueba-por-P-resolucion
    '(((- p) q) ((- q) r) ((- r) s) (r) ((- s))))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {-R,S}
4 NIL {R}
5 NIL {-S}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {-R,S}
4 NIL {R}
  ** 6 (4 3) {S}
7 (6 5) {}

========== Prueba ===========
3 NIL {-R,S}
4 NIL {R}
5 NIL {-S}
6 (4 3) {S}
7 (6 5) {}
===== Fin de la prueba ======
T
```

N–resolución

- Cláusulas negativas
  - C es negativa si todos sus literales son negativos
  - Implementación: **(es-ca-negativa Ca)**
  - Ejemplos:

```
> (setf ca1 (crea-clausula-anotada :numero 1
                                   :clausula '((- p) (- q) (- r))))
1 NIL {-P,-Q,-R}
> (es-ca-negativa ca1)
T
> (setf ca2 (crea-clausula-anotada :numero 1 :clausula '(p (- q))))
2 NIL {P,-Q}
> (es-ca-negativa ca2)
NIL
```


N–resolución

- Prueba por N–resolución
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por N–resolución de C a partir de S si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones:
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$ y alguna de las cláusulas C_j o C_k es negativa
  - C es probable por N–resolución a partir de S si existe una prueba por N–resolución de C a partir de S
  - Representación: $S \vdash_N C$
- Propiedades
  - $S \vdash_N \{\} \iff S$ es inconsistente

Estrategia del conjunto soporte

- Prueba por resolución con soporte
  - Sea S un conjunto de cláusulas y $T \subseteq S$ con $S - T$ consistente
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por resolución de C a partir de S con soporte T si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$ y alguna de las cláusulas C_j o C_k no pertenece a $S - T$
  - C es probable por resolución con soporte T a partir de S si existe una prueba por resolución de C a partir de S con soporte T
  - Representación: $S \vdash_{\text{soporte } T} C$
- Propiedades
  - Si $T \subseteq S$ y $S - T$ es consistente, entonces S es inconsistente $\iff S \vdash_{\text{soporte } T} \{\}$

Estrategia del conjunto soporte

- Búsqueda de refutaciones

```
> (prueba-por-resolucion-con-soporte
   '(((- p) q) ((- q) r) ((- r) s) (p))
   '(((- q))))

========== Usables ==========
1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {-R,S}
4 NIL {P}
====== Fin de usables =======

========== Soporte ==========
5 NIL {-Q}
====== Fin del soporte ======

5 NIL {-Q}
  ** 6 (5 1) {-P}
7 (6 4) {}

========== Prueba ===========
1 NIL {-P,Q}
4 NIL {P}
5 NIL {-Q}
6 (5 1) {-P}
7 (6 4) {}
===== Fin de la prueba ======
T
```

Resolución semántica

- Prueba por resolución semántica
  - Se considera una interpretación I
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por resolución semántica de C a partir de S si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones:
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$ y alguna de las cláusulas C_j o C_k es falsa en I
  - C es probable por resolución semántica a partir de S si existe una prueba por resolución semántica de C a partir de S
  - Representación: $S \vdash_I C$
- Propiedades
  - $S \vdash_I \{\} \iff S$ es inconsistente

Resolución semántica

- Resolución positiva = Resolución semántica con $I = \{\}$
- Resolución negativa = Resolución semántica con $I =$ SP
- Resolución en S con soporte T = Resolución semántica con I un modelo de $S - T$

Resolución unidad

- Prueba por resolución unidad
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por resolución unidad de C a partir de S si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones:
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$ y alguna de las claúsulas C_j o C_k es unitaria
  - C es probable por resolución unidad a partir de S si existe una prueba por resolución unidad de C a partir de S
  - Representación: $S \vdash_U C$
- Propiedades
  - Adecuación: $S \vdash_U \{\} \Longrightarrow S$ es inconsistente
  - S es inconsistente $\not\Longrightarrow S \vdash_U \{\}$
    - Sea $S = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}$: S es inconsistente, pero $S \not\vdash_U \{\}$

Resolución unidad

- Búsqueda de refutaciones

```
> (prueba-por-resolucion-unidad
   '((p q) ((- p) r) ((- q) r) ((- q) s) ((- r))))

========== Soporte ==========
1 NIL {P,Q}
2 NIL {-P,R}
3 NIL {-Q,R}
4 NIL {-Q,S}
5 NIL {-R}
====== Fin del soporte ======

1 NIL {P,Q}
2 NIL {-P,R}
3 NIL {-Q,R}
4 NIL {-Q,S}
5 NIL {-R}
  ** 6 (5 2) {-P}
  ** 7 (5 3) {-Q}
6 (5 2) {-P}
  ** 8 (6 1) {Q}
9 (8 7) {}

========== Prueba ===========
1 NIL {P,Q}
2 NIL {-P,R}
3 NIL {-Q,R}
5 NIL {-R}
6 (5 2) {-P}
7 (5 3) {-Q}
8 (6 1) {Q}
9 (8 7) {}
===== Fin de la prueba ======
T
```

Resolución por entradas

- Prueba por resolución por entradas
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por resolución por entradas de C a partir de S si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones:
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i \in resolventes(C_j, C_k)$ y alguna de las cláusulas C_j o C_k pertenece a S
  - C es probable por resolución por entradas a partir de S si existe una prueba por resolución por entradas de C a partir de S
  - Representación: $S \vdash_E C$
- Propiedades
  - Adecuación: $S \vdash_E \{\} \Longrightarrow S$ es inconsistente
  - S es inconsistente $\not\Longrightarrow S \vdash_E \{\}$
    - Sea $S = \{\{p, q\}, \{p, \neg q\}, \{\neg p, q\}, \{\neg p, \neg q\}\}$: S es inconsistente, pero $S \not\vdash_E \{\}$

Resolución por entradas

- Búsqueda de refutaciones

```
> (prueba-por-resolucion-por-entradas
   '((p q) ((- p) r) ((- q) r) ((- q) s) ((- r))))

========== Soporte ==========
1 NIL {P,Q}
2 NIL {-P,R}
3 NIL {-Q,R}
4 NIL {-Q,S}
5 NIL {-R}
====== Fin del soporte ======

1 NIL {P,Q}
  ** 6 (1 2) {Q,R}
  ** 7 (1 3) {P,R}
  ** 8 (1 4) {P,S}
2 NIL {-P,R}
  ** 9 (2 5) {-P}
3 NIL {-Q,R}
  ** 10 (3 5) {-Q}
4 NIL {-Q,S}
5 NIL {-R}
6 (1 2) {Q,R}
  ** 11 (6 3) {R}
12 (11 5) {}

========== Prueba ===========
1 NIL {P,Q}
2 NIL {-P,R}
3 NIL {-Q,R}
5 NIL {-R}
6 (1 2) {Q,R}
11 (6 3) {R}
12 (11 5) {}
===== Fin de la prueba ======
T
```

Resolución ordenada

- Prueba por resolución ordenada
  - Se considera un orden entre los símbolos proposicionales
  - La sucesión $(C_1, \ldots, C_n)$ es una prueba por resolución ordenada de C a partir de S si $C_n = C$ y para todo $i \in \{1, \ldots, n\}$ se verifica una de las siguientes condiciones:
    - $C_i \in S$;
    - existen $j, k < i$ tales que $C_i = resolvente(C_j, C_k, L)$ y el símbolo de L es el mayor dentro de C_j y dentro de C_k
  - C es probable por resolución ordenada a partir de S si existe una prueba por resolución ordenada de C a partir de S
  - Representación: $S \vdash_< C$
- Propiedades
  - $S \vdash_< \{\} \iff S$ es inconsistente

Resolución ordenada

- Búsqueda de refutaciones

```
> (prueba-por-resolucion-ordenada
   '((p q) ((- p) r) ((- q) r) ((- q) s) ((- r))))

========== Soporte ==========
1 NIL {Q,P}
2 NIL {R,-P}
3 NIL {R,-Q}
4 NIL {S,-Q}
5 NIL {-R}
====== Fin del soporte ======

1 NIL {Q,P}
2 NIL {R,-P}
3 NIL {R,-Q}
4 NIL {S,-Q}
5 NIL {-R}
  ** 6 (5 3) {-Q}
  ** 7 (5 2) {-P}
6 (5 3) {-Q}
  ** 8 (6 1) {P}
9 (8 7) {}

========== Prueba ===========
1 NIL {Q,P}
2 NIL {R,-P}
3 NIL {R,-Q}
5 NIL {-R}
6 (5 3) {-Q}
7 (5 2) {-P}
8 (6 1) {P}
9 (8 7) {}
===== Fin de la prueba ======
T
```

Resolución lineal

- Prueba por resolución lineal
  - La sucesión $(C_0, \ldots, C_n)$ es una demostración por resolución lineal de C a partir de S con base $C_0 \in S$ si $C_n = C$ y para cada $i \in \{1, \ldots, n\}$ existe un $B_{i-1} \in S \cup \{C_0, \ldots, C_{i-1}\}$ tal que $C_i \in resolventes(C_{i-1}, B_{i-1})$
    - C_0 es la cláusula base
    - C_i se llaman cláusulas centrales
    - B_i se llaman cláusulas laterales
  - C es probable por resolución lineal a partir de S con base C_0 si existe una prueba por resolución lineal de C a partir de S con base C_0
  - Representación: $S \vdash_{L,C_0} C$
- Propiedades
  - Si $C_0 \in S$ y $S - \{C_0\}$ es consistente, entonces S es inconsistente $\iff S \vdash_{L,C_0} \{\}$

Resolución lineal

- Búsqueda de refutaciones

```
> (prueba-por-resolucion-lineal
   '((p (- q)) (p (- r)) (r))
   '((- p)))

========= Entradas ==========
1 NIL {P,-Q}
2 NIL {P,-R}
3 NIL {R}
===== Fin de entradas =======

4 NIL {-P}
5 (4 1) {-Q}
··· Fallo ···

6 (4 2) {-R}
7 (6 3) {}

========== Prueba ===========
2 NIL {P,-R}
3 NIL {R}
4 NIL {-P}
6 (4 2) {-R}
7 (6 3) {}
===== Fin de la prueba ======
T
```

Ejercicios

- *Consideremos el siguiente conjunto de cláusulas $S = \{p \vee q, q \vee r, r \vee w, \neg r \vee \neg p, \neg w \vee \neg q, \neg q \vee \neg r\}$. Demostrar que es insatisfacible mediante*
  - *Resolución positiva*
  - *Resolución por entradas*
  - *Resolución ordenada con el orden $p > q > r > s$*
- *Consideremos el siguiente conjunto de cláusulas $S = \{\neg p \vee \neg r \vee \neg s, \neg p \vee q \vee s, p \vee \neg r, \neg q \vee r, \neg q \vee \neg r \vee s, q \vee r\}$. Demostrar que es insatisfacible mediante*
  - *Resolución negativa*
  - *Resolución lineal (escoger libremente la cláusula inicial)*
  - *Resolución con conjunto soporte (escoger libremente el conjunto T)*

Bibliografía

- Chang, C.L. y Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving*. (Academic Press, 1973)
  - Cap. 5: “The resolution principle”
- Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).
  - Cap. 2: “Logic and resolution”
- Genesereth, M.R. y Nilsson, N.J. *Logical Foundations of Artificial Intelligence* (Morgan Kaufmann, 1987)
  - Cap. 4: “Resolution”
- Russell, S. y Norvig, P. *Inteligencia artificial (un enfoque moderno)* (Prentice–Hall, 1996)
  - Cap. 9.5: “Resolución: un procedimiento completo de inferencia”

Capítulo 11

Programación lógica proposicional

Lógica y Programación

Programación lógica proposicional

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Secuentes

- Sintaxis de secuente
  - $p_1, \ldots, p_m \leftarrow q_1, \ldots, q_n$ con $n \geq 0$, $m \geq 0$, p_i y q_j símbolos
  - Cabeza del secuente: $\{p_1, \ldots, p_m\}$
  - Cuerpo del secuente: $\{q_1, \ldots, q_n\}$
  - Implementación:

```
(defun cabeza (S)
  (first S))

(defun cuerpo (S)
  (third S))

(defun crea-secuente (Lp Lq)
  (list Lp '<- Lq))
```

  - Ejemplos

```
> (crea-secuente '(p) '(q r))
((P) <- (Q R))
> (cabeza '((p) <- (q r)))
(P)
> (cuerpo '((p) <- (q r)))
(Q R)
```

Secuentes

- Relación entre secuentes, fórmulas y cláusulas
  - Secuente: $p_1, \ldots, p_m \leftarrow q_1, \ldots, q_n$
  - Fórmula: $q_1 \wedge \ldots \wedge q_n \rightarrow p_1 \vee \ldots \vee p_m$
  - Cláusula: $\{p_1, \ldots, p_m, \neg q_1, \ldots, \neg q_n\}$
- Transformación de secuentes a cláusulas
  - Implementación: **`(secuente->clausula` S`)`**
  - Ejemplos:

```
> (secuente->clausula '((a) <- (b c)))
(A (- B) (- C))
> (secuente->clausula '((a) <- ()))
(A)
```


Cláusulas de Horn

- Cláusulas de Horn
  - Un secuente representa una cláusula de Horn si su cabeza tiene como máximo un elemento
  - Cláusulas de Horn son las que no tienen más de un literal positivo
- Propiedades
  - Completitud de resolución unidad para cláusulas de Horn
  - Completitud de resolución por entradas para cláusulas de Horn
- Clasificación de cláusulas de Horn:
  - Cláusula definida:
    - Regla: $p \leftarrow q_1, \ldots, q_n$ con $n > 0$
    - Hecho: $p \leftarrow$
  - Objetivo definido:
    - Objetivo propio: $\leftarrow q_1, \ldots, q_n$ con $n > 0$
    - Objetivo vacío: $\leftarrow$

Sintaxis de programas lógicos

- Un programa lógico es un conjunto finito de cláusulas definidas
  - Ejemplo de programa lógico

```
p  <- q, r
p  <- s
r  <-
q  <-
p1 <- s
p2 <- s
```

  - Representación

```
( ((p)  <- (q r))
  ((p)  <- (s))
  ((r)  <- ())
  ((q)  <- ())
  ((p1) <- (s))
  ((p2) <- (s))   )
```


Semántica declarativa

- Base de Herbrand:
  - La base de Herbrand, $BH(P)$, de un programa lógico, P, es el conjunto de los símbolos proposicionales que aparecen en el programa
  - Implementación: **`(base-de-Herbrand`** P**`)`**
  - Ejemplos:

```
> (base-de-Herbrand
    '(((p)  <- (q r))
      ((p)  <- (s))
      ((r)  <- ())
      ((q)  <- ())
      ((p1) <- (s))
      ((p2) <- (s))))
(P Q R S P1 P2)
```

Semántica declarativa

- Modelos de Herbrand
  - Los modelos de Herbrand de un programa lógico P son aquellos modelos que están contenidos en la base de Herbrand de P
  - Implementación: **(modelos-de-Herbrand P)**
  - Ejemplos:

```
> (modelos-de-Herbrand
    '(((p)  <- (q r))
      ((p)  <- (s))
      ((r)  <- ())
      ((q)  <- ())
      ((p1) <- (s))
      ((p2) <- (s))))
((P Q R) (P Q R P2) (P Q R P1) (P Q R P1 P2) (P Q R S P1 P2))
```


Semántica declarativa

- Propiedades: Sea P un programa lógico
  - $BH(P) \models P$
  - La intersección de modelos de P es modelo de P
- Menor modelo de Herbrand
  - I es el menor modelo de Herbrand de P si I es un modelo de P y es subconjunto de todos los modelos de P
  - MMH$(P) = \bigcap\{I : I \models P\}$
  - MMH$(P) = \{p : P \models p\}$
  - Implementación: **(menor-modelo-de-Herbrand P)**
  - Ejemplos:

```
> (menor-modelo-de-Herbrand
    '(((p)  <- (q r)) ((p)  <- (s)) ((r)  <- ())
      ((q)  <- ())    ((p1) <- (s)) ((p2) <- (s))))
(P Q R)
```

Semántica de punto fijo

- Operador de consecuencia inmediata
  $T_P(I) = \{p : (p \leftarrow q_1, \ldots, q_n) \in P \text{ y } \{q_1, \ldots, q_n\} \subseteq I\}$
- Ejemplo:
  - Programa P:
    ```
    p  <- q, r    p  <- s
    r  <-         q  <-
    p1 <- s       p2 <- s
    ```
  - Aplicación del operador de consecuencia inmediata:
    $T_P(\emptyset) = \{r, q\}$
    $T_P(\{r, q\}) = \{p, r, q\}$
    $T_P(\{p, r, q\}) = \{p, r, q\}$
  - Menor punto fijo: MPF$(P) = \{p, r, q\}$
- Propiedad: MPF(P) = MMH(P)
- Encadenamiento adelante

Semántica de punto fijo

- Consecuencias inmediatas
  - Implementación: `(consecuencias-inmediatas P)`
  - Ejemplo
    ```
    > (setf *Prog* '(((p)  <- (q r)) ((p)  <- (s)) ((r)  <- ())
                     ((q)  <- ())    ((p1) <- (s)) ((p2) <- (s))))
    > (consecuencias-inmediatas *Prog* ())
      (R Q)
    > (consecuencias-inmediatas *Prog* '(r q))
      (P R Q)
    > (consecuencias-inmediatas *Prog* '(p r q))
      (P R Q)
    ```
- Menor punto fijo
  - Implementación: `(menor-punto-fijo P)`
  - Ejemplo:
    ```
    > (menor-punto-fijo *Prog*)
    (P R Q)
    ```

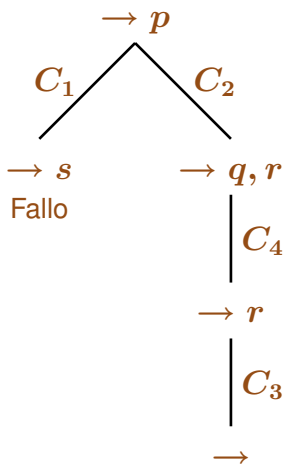
Resolución SLD

- SLD–resolución: Resolución **L**ineal con función de **S**elección para cláusulas **D**efinidas
- SLD–resolventes
  - Objetivo: $\leftarrow p_1, \ldots, p_{i-i}, p_i, p_{i+1}, \ldots, p_n$
  - Cláusula: $p_i \leftarrow q_1, \ldots, q_m$
  - SLD–resolvente: $\leftarrow p_1, \ldots, p_{i-i}, q_1, \ldots, q_m, p_{i+1}, \ldots, p_n$
- Doble indeterminismo
  - Regla de computación: Selección del literal del objetivo
  - Regla de elección: Selección de la cláusula del programa

Arbol de SLD–resolución

- Ejemplo 1

C_1: $p \rightarrow s$
C_2: $p \rightarrow q, r$
C_3: $r \rightarrow$
C_4: $q \rightarrow$
C_5: $p1 \rightarrow s$
C_6: $p2 \rightarrow s$

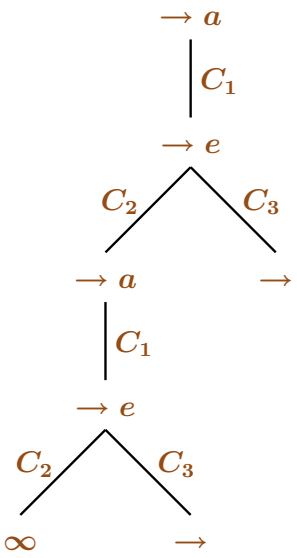


Arbol de SLD–resolución

- Ejemplo 2

C_1: $a \longrightarrow e$
C_2: $e \longrightarrow a$
C_3: $e \longrightarrow$

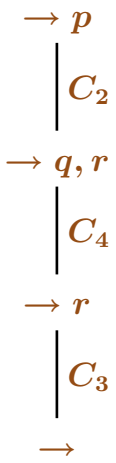
- Ramas:
  - éxito
  - fallo
  - infinita
- Estrategias de búsqueda:
  - profundidad
  - anchura



SLD–resolución

- Ejemplo 1

C_1: $p \longrightarrow s$
C_2: $p \longrightarrow q, r$
C_3: $r \longrightarrow$
C_4: $q \longrightarrow$
C_5: $p1 \longrightarrow s$
C_6: $p2 \longrightarrow s$



SLD–Resolución

- Ejemplo 2

C_1: $a \rightarrow b, c$
C_2: $b \rightarrow c, d$
C_3: $c \rightarrow e$
C_4: $d \rightarrow$
C_5: $e \rightarrow$

$\rightarrow a$
$\quad | C_1$
$\rightarrow b, c$
$\quad | C_2$
$\rightarrow c, d, c$
$\quad | C_3$
$\rightarrow e, d, c$
$\quad | C_5$
$\rightarrow d, c$
$\quad | C_4$
$\rightarrow c$
$\quad | C_3$
$\rightarrow e$
$\quad | C_5$
$\rightarrow$

Resolución SLD

- SLD–resolución: resolución lineal por entradas
- Completitud de SLD–resolución
  - P un programa lógico
  - $G = \leftarrow p_1, \ldots, p_n$ un objetivo
  - Son equivalentes:
    - $P \cup \{G\} \vdash_{SLD} \{\}$
    - $P \cup \{G\}$ es inconsistente
    - $P \models p_1 \wedge \ldots \wedge p_n$

Procedimiento de SLD–resolución

- Características:
  - Elección de literal en objetivo: El primero
  - Elección de la cláusula: La primera que se equipare
  - Estrategia de búsqueda: En profundidad
- Implementación: **(prueba-por-SLD-resolucion** P G**)**

Procedimiento de SLD–resolución

- Traza

```
> (setf *programa*
    '(((p)  <- (s)) ((p)  <- (q r)) ((r)  <- ())
      ((q)  <- ())  ((p1) <- (s))   ((p2) <- (s))))
> (trace SLD-resolvente)
(SLD-RESOLVENTE)
> (prueba-por-SLD-resolucion
    *programa* '(() <- (p)))
(SLD-RESOLVENTE '(NIL <- (P)) '((P) <- (S)))
SLD-RESOLVENTE ==> (NIL <- (S))
(SLD-RESOLVENTE '(NIL <- (P)) '((P) <- (Q R)))
SLD-RESOLVENTE ==> (NIL <- (Q R))
(SLD-RESOLVENTE '(NIL <- (Q R)) '((Q) <- NIL))
SLD-RESOLVENTE ==> (NIL <- (R))
(SLD-RESOLVENTE '(NIL <- (R)) '((R) <- NIL))
SLD-RESOLVENTE ==> (NIL <- NIL)
T
```

Procedimiento de SLD–resolución

```
(defun prueba-por-SLD-resolucion (P G)
  (if (es-objetivo-vacio G)
      t
      (some #'(lambda (C)
                (and (eql (first (cabeza C))
                          (first (cuerpo G)))
                     (prueba-por-SLD-resolucion
                        P
                        (SLD-resolvente G C))))
            P)))

(defun es-objetivo-vacio (G)
  (and (null (cabeza G))
       (null (cuerpo G))))

(defun SLD-resolvente (G C)
  (crea-secuente () (append (cuerpo C) (rest (cuerpo G)))))
```


Incompletitud de Prolog

- Traza

```
> (trace SLD-resolvente)
(SLD-RESOLVENTE)
> (prueba-por-SLD-resolucion
    '(((aprueba) <- (estudia))
      ((estudia) <- (aprueba))
      ((estudia) <- ()))
    '(() <- (aprueba)))
(SLD-RESOLVENTE '(NIL <- (APRUEBA))
                '((APRUEBA) <- (ESTUDIA)))
SLD-RESOLVENTE ==> (NIL <- (ESTUDIA))
(SLD-RESOLVENTE '(NIL <- (ESTUDIA))
                '((ESTUDIA) <- (APRUEBA)))
SLD-RESOLVENTE ==> (NIL <- (APRUEBA))
(SLD-RESOLVENTE '(NIL <- (APRUEBA))
                '((APRUEBA) <- (ESTUDIA)))
...
```

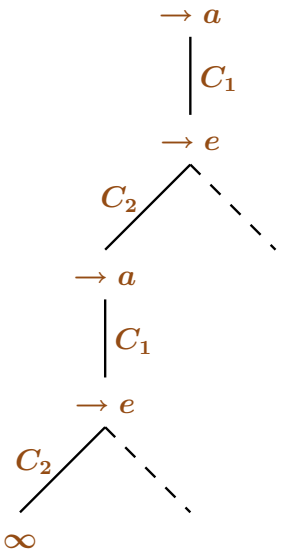
Incompletitud de Prolog

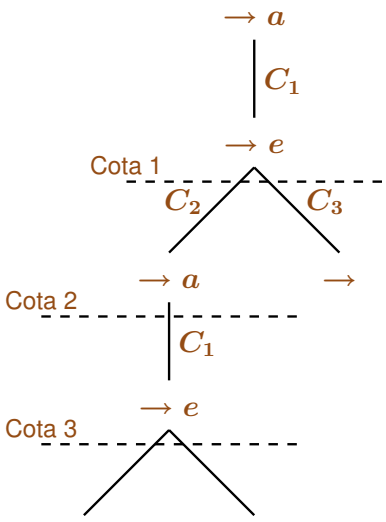
- Traza

```
> (prueba-por-SLD-resolucion
     '(((estudia) <- ())
       ((aprueba) <- (estudia))
       ((estudia) <- (aprueba)))
     '(() <- (aprueba)))
(SLD-RESOLVENTE '(NIL <- (APRUEBA))
                '((APRUEBA) <- (ESTUDIA)))
SLD-RESOLVENTE ==> (NIL <- (ESTUDIA))
(SLD-RESOLVENTE '(NIL <- (ESTUDIA))
                '((ESTUDIA) <- NIL))
SLD-RESOLVENTE ==> (NIL <- NIL)
T
```


SLD–resolución acotada

- Problema: ramas infinitas
- Búsqueda en profundidad acotada





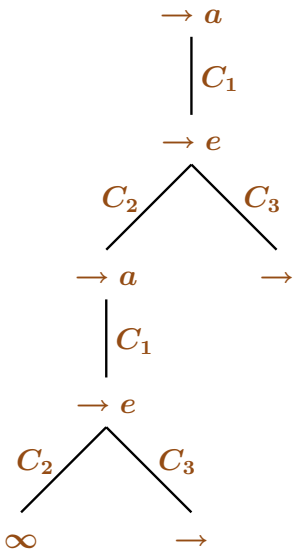
SLD–resolución acotada

- Implementación: **(prueba-por-SLD-resolucion** P G $cota$**)**
- Ejemplos:

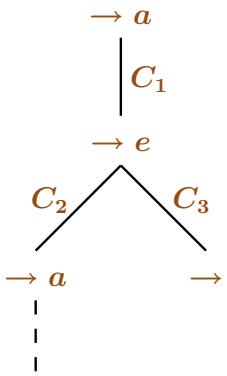
```
> (defvar *Programa-con-rama-infinita*
    '(((a) <- (e))
      ((e) <- (a))
      ((e) <- ())))
> (prueba-por-SLD-resolucion *Programa-con-rama-infinita*
                             '(() <- (a)))
*** - Lisp stack overflow. RESET
> (prueba-por-SLD-resolucion-acotada *Programa-con-rama-infinita*
                                     '(() <- (a))
                                     1)
NIL
> (prueba-por-SLD-resolucion-acotada *Programa-con-rama-infinita*
                                     '(() <- (a))
                                     2)
T
```


SLD–resolución en anchura

- Problema: ramas infinitas



- Búsqueda en anchura



SLD–resolución en anchura

- Implementación: **(prueba-por-SLD-resolucion-anchura P G)**
- Ejemplo:

```
> (prueba-por-SLD-resolucion *Programa-con-rama-infinita*
                             '(() <- (a)))
*** - Lisp stack overflow. RESET

> (prueba-por-SLD-resolucion-anchura *Programa-con-rama-infinita*
                                     '(() <- (a)))
T
```


SLD–resolución con objetivos resueltos

- Objetivos duplicados

C_1: $p \longrightarrow q, r$
C_2: $q \longrightarrow s$
C_3: $r \longrightarrow q$
C_4: $s \longrightarrow$

$\longrightarrow p$
$\quad | C_1$
$\longrightarrow q, r$
$\quad | C_2$
$\longrightarrow s, r$
$\quad | C_4$
$\longrightarrow r$
$\quad | C_3$
$\longrightarrow q$
$\quad | C_2$
$\longrightarrow s$
$\quad | C_4$
$\longrightarrow$

- Con objetivos resueltos

$\longrightarrow p \qquad \{\}$
$\quad | C_1$
$\longrightarrow q, r \qquad \{\}$
$\quad | C_2$
$\longrightarrow s, r \qquad \{\}$
$\quad | C_4$
$\longrightarrow r \qquad \{s, q\}$
$\quad | C_3$
$\longrightarrow q \qquad \{s, q\}$

$\longrightarrow$

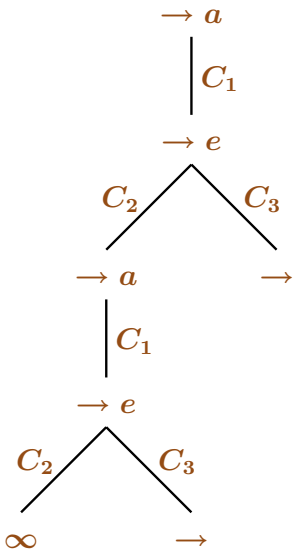
SLD–resolución con objetivos resueltos

- Implementación: **(prueba-por-SLD-resolucion-resueltos P G)**
- Ejemplo:

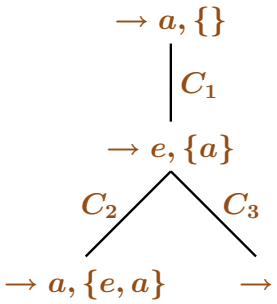
```
> (prueba-por-SLD-resolucion-resueltos *P* '(nil <- (p)))
  (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- (P)) '(NIL))
    (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- (Q R)) '(NIL))
      (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- (Q)) '(NIL))
        (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- (S)) '(NIL))
          (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- NIL) '(NIL))
          ==> (NIL)
        ==> ((S))
      ==> ((Q S))
      (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- (R)) '((Q S)))
        (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- (Q)) '((Q S)))
          (PRUEBA-POR-SLD-RESOLUCION-RESUELTOS *P* '(NIL <- NIL) '((Q S)))
          ==> ((Q S))
        ==> ((Q S))
      ==> ((R Q S))
    ==> ((R Q S))
  ==> ((P R Q S))
((P R Q S))
```


SLD–resolución con objetivos pendientes

- Problema: ramas infinitas



- Objetivos pendientes



SLD–resolución con objetivos resueltos

- Implementación: **(prueba-por-SLD-resolucion-pendientes P G)**
- Ejemplo:

```
> (PRUEBA-POR-SLD-RESOLUCION-PENDIENTES *P* '(NIL <- (A)))
    (PRUEBA-POR-SLD-RESOLUCION-PENDIENTES *P* '(NIL <- (E)) '(A))
      (PRUEBA-POR-SLD-RESOLUCION-PENDIENTES *P* '(NIL <- (A)) '(E A))
      ==> NIL
      (PRUEBA-POR-SLD-RESOLUCION-PENDIENTES *P* '(NIL <- NIL) '(E A))
      ==> T
    ==> T
  ==> T
T
```


Referencias

- Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).
  - Cap. 2: “Logic and resolution”
- Rich, E. y Knight, K. *Inteligencia artificial (segunda edición)* (McGraw–Hill Interamericana, 1994).
  - Cap. 5: “La lógica de predicados”
- Russell, S. y Norvig, P. *Inteligencia artificial (un enfoque moderno)* (Prentice–Hall, 1996).
  - Cap. 9: “La inferencia en la lógica de primer orden”
  - Cap. 10: “Sistemas de razonamiento lógico”
- Winston, P.R. *Inteligencia Artificial (3a. ed.)* (Addison–Wesley, 1994).
  - Cap. 13: “Lógica y prueba de resolución”

Capítulo 12

Programación lógica y Prolog

Lógica y Programación

Programación lógica y Prolog

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Necesidad de ampliación

- Lenguaje natural
  - Javier defiende a todos sus amigos.
  - Fernando es un amigo de Javier
  - Por tanto, Javier defiende a Fernando
- Lógica relacional
  - $F_1 : (\forall A)[amigo\_de(A, javier) \rightarrow defiende(javier, A)]$
  - $F_2 : amigo\_de(fernando, javier)$
  - $F_3 : defiende(javier, fernando)$
  - $\{F_1, F_2\} \models F_3$
- Programación lógica relacional
  - $C_1 : defiende(javier, A) \leftarrow amigo\_de(A, javier)$
  - $C_2 : amigo\_de(fernando, javier) \leftarrow$
  - $C_3 : defiende(javier, fernando) \leftarrow$
  - $\{C_1, C_2\} \models C_3$

Programación lógica relacional

- Gramática

Constante	::= <palabra comenzando con minúscula>
Variable	::= <palabra comenzando con mayúscula>
Término	::= *Constante* | *Variable*
Predicado	::= <palabra comenzando con minúscula>
Átomo	::= *Predicado*[(*Término*[,*Término*]*)]
Expresión	::= *Término* | *Átomo*
Cabeza	::= [*Átomo*]
Cuerpo	::= [*Átomo*[,*Átomo*]*]
CláusulaDefinida	::= *Cabeza* <- *Cuerpo*

- Programa lógico relacional: Conjunto de cláusulas definidas
- Ejemplo (`*P1*`)

```
defiende(javier,A) <- amigo_de(A,javier)
amigo_de(fernando,javier) <-
```

- Expresiones básicas: sin variables

Programación lógica relacional

- Universo de Herbrand $UH(P)$
  - Conjunto de términos básicos de P
  - $UH($`*P1*`$) = \{$`javier, fernando`$\}$
- Base de Herbrand $BH(P)$
  - Conjunto de átomos construidos con predicados de P y términos básicos de P
  - $BH($`*P1*`$)$

```
{defiende(javier,javier),
 defiende(javier,fernando),
 defiende(fernando,javier),
 defiende(fernando,fernando),
 amigo_de(javier,javier),
 amigo_de(javier,fernando),
 amigo_de(fernando,javier),
 amigo_de(fernando,fernando)}
```

Programación lógica relacional

- Interpretación de Herbrand
  - I interpretación de Herbrand de P si $I \subseteq BH(P)$
  - Una interpretación de Herbrand de `*P1*` es
    ```
    {defiende(javier,fernando),
      amigo_de(fernando,javier)}
    ```
  - `*P1*` tiene $2^8 = 256$ interpretaciones de Herbrand

Sustituciones

- Una sustitución es una aplicación de las variables en los términos
  - $\sigma = \{A/fernando\}$
- Aplicación de una sustitución a una cláusula
  - $C\sigma$: sustituir cada variable X de C por $\sigma(X)$
  - Ejemplo:

    $C \quad = defiende(javier, A1) \leftarrow amigo\_de(A, javier).$

    $\sigma \quad = \{A/fernando\}$

    $C\sigma \quad = defiende(javier, A1) \leftarrow amigo\_de(fernando, javier)$

Instancias

- C es instancia de D si existe σ tal que $C = D\sigma$
- Instancia básicas
- Las instancias básicas de **`*P1*`** son:

```
defiende(javier,javier) <- amigo_de(javier,javier)
defiende(javier,fernando) <- amigo_de(fernando,javier)
amigo_de(fernando,javier) <-
```

- Programa proposicional correspondiente al básico (**`*P1b*`**):

```
d-j-j  <- ad-j-j
d-j-f  <- ad-f-j
ad-f-j <-
```


Modelos de Herbrand

- $I \models C$ si I es modelo de todas las instancias básicas de C
- $I \models P$ si I es modelo de todas las cláusulas de P
- Ejemplo

```
> (modelos-de-Herbrand *P1b*)
((D-J-F AD-F-J)
 (D-J-F D-J-J AD-F-J)
 (D-J-F AD-J-J D-J-J AD-F-J))
> (menor-modelo-de-Herbrand *P1b*)
(D-J-F AD-F-J)
```

Semántica de punto fijo

- Programa básico $basico(P) = \{C\sigma : C \in P$ y $C\sigma$ es básica$\}$
- Operador de consecuencia inmediata
  $T_P(I) = \{A : (A \leftarrow B_1, \ldots, B_n) \in basico(P)$ y
  $\{B_1, \ldots, B_n\} \subseteq I\}$
- Consecuencias
  $I_0 = \emptyset$
  $I_1 = T_{\texttt{*P1*}}(I_0) = \{amigo\_de(fernando, javier)\}$
  $I_2 = T_{\texttt{*P1*}}(I_1) = I_1 \cup \{defiende(javier, fernando)\}$
  $I_3 = T_{\texttt{*P1*}}(I_2) = I_2$
- Menor punto fijo
  MPF(`*P1*`) $= \{amigo\_de(fernando, javier),$
  $defiende(javier, fernando)\}$
- MPF(P) = MMH(P)

Necesidad de ampliación

- Lenguaje natural: “Cada persona tiene un padre”
- Programación lógica relacional:

```
es_padre_de(adan,cain)
es_padre_de(adan,abel)
...
```

- Lógica de primer orden: $(\forall X)(\exists Y)$[es_padre_de(Y, X)]
- Programación lógica:

```
es_padre_de(el_padre_de(X),X) <-
```

Necesidad de ampliación

- Lenguaje natural:
  - "El cero es un número entero"
  - "Todo número entero tiene un siguiente"
- Lógica de primer orden:
  - entero$(\mathbf{0})$
  - $(\forall \boldsymbol{X})$[entero$(\boldsymbol{X}) \rightarrow (\exists \boldsymbol{Y})$[es_siguiente$(\boldsymbol{Y}, \boldsymbol{X})$]]
- Programación lógica:

```
entero(0) <-
es_siguiente(s(X),X) <- entero(X)
```


Programación lógica

- Gramática

Functor	::= <palabra comenzando con minúscula>
Variable	::= <palabra comenzando con mayúscula>
Término	::= *Variable* | *Functor*[(*Término*[,*Término*]*)]
Predicado	::= <palabra comenzando con minúscula>
Átomo	::= *Predicado*[(*Término*[,*Término*]*)]
Expresión	::= *Término* | *Átomo*
Cabeza	::= [*Átomo*]
Cuerpo	::= [*Átomo*[,*Átomo*]*]
CláusulaDefinida	::= *Cabeza* <- *Cuerpo*

Programación lógica

- Programa lógico: Conjunto de cláusulas definidas
- Ejemplo (`*P2*`)

```
suma(0,X,X) <-
suma(s(X),Y,s(Z)) <- suma(X,Y,Z)
```

- Significado

  $(\forall X)[suma(0, X, X)]$

  $(\forall X, Y, Z)[suma(X, Y, Z) \rightarrow suma(s(X), Y, s(Z))]$

Programación lógica

- Universo de Herbrand $UH(P)$
  - Conjunto de términos básicos de P
  - $UH($`*P2*`$) = \{0, s(0), s(s(0)), \ldots\}$
- Base de Herbrand $BH(P)$
  - Conjunto de átomos construidos con predicados de P y términos básicos de P
  - $BH($`*P2*`$)$

```
{suma(0,0,0), suma(0,0,s(0)), suma(0,0,s(s(0))), ...
 suma(0,s(0),0), suma(0,s(0),s(0)), ...}
```

Programación lógica

- Interpretaciones de Herbrand
  - I interpretación de Herbrand de P si $I \subseteq BH(P)$
- Modelos de Herbrand
  - $I \models C$ si I es modelo de todas las instancias básicas de C
  - $I \models P$ si I es modelo de todas las cláusulas de P
  - Ejemplos de modelos de **`*P2*`**:

```
I1 = BH(*P2*)
I2 = {suma(0,0,0), suma(0,s(0),s(0)), ...
      suma(s(0),0,s(0)), suma(s(0),s(0),s(s(0))), ...}
```

- Menor modelo de Herbrand de P

Semántica de punto fijo

- Programa básico $basico(P) = \{C\sigma : C \in P$ y $C\sigma$ es básica$\}$
- Operador de consecuencia inmediata:

  $T_P(I) = \{A : (A \leftarrow B_1, \ldots, B_n) \in basico(P)$ y
  $\{B_1, \ldots, B_n\} \subseteq I\}$
- Consecuencias

  $I_0 = \emptyset$
  $I_1 = T_{\texttt{*P2*}}(I_0) = I_0 \cup \{suma(0,0,0), suma(0,s(0),s(0)), \ldots\}$
  $I_2 = T_{\texttt{*P2*}}(I_1) = I_1 \cup \{\ suma(s(0),0,s(0)),$
  $suma(s(0),s(0),s(s(0))), \ldots\}$
  $I_3 = T_{\texttt{*P2*}}(I_2) = I_2 \cup \{\ suma(s(s(0)),0,s(s(0))),$
  $suma(s(s(0)),s(0),s(s(s(0)))), \ldots\}$
- Menor punto fijo

  MPF(**`*P2*`**) $= \{\ suma(0,0,0), suma(0,s(0),s(0)), \ldots,$
  $suma(s(0),0,s(0)), suma(s(0),s(0),s(s(0))), \ldots\}$
- MPF(P) = MMH(P)

Unificación

- Notación
  - Variables: $X, X_1, X_2, \ldots, Y, Y_1, Y_2, \ldots$
  - Constantes: $a, a_1, a_2, \ldots, b, b_1, b_2, \ldots$
  - Símbolos de función: $f, f_1, f_2, \ldots, g, g_1, g_2, \ldots$
  - Símbolos de predicados: $p, p_1, p_2, \ldots, q, q_1, q_2, \ldots$
  - Términos: $t, t_1, t_2, \ldots, s, s_1, s_2, \ldots$
  - Fórmulas atómicas: $A, A_1, A_2, \ldots, B, B_1, B_2, \ldots$
  - Expresiones: $E, E_1, E_2, \ldots$
  - Sustituciones: $\epsilon, \sigma, \sigma_1, \sigma_2, \ldots, \theta, \theta_1, \theta_2, \ldots$

Unificación

- Representación de variables en Lisp: Simbolos que comienzan por `x`, `y`, `z`, `u`, `v` o `w`
- Implementación:

```
(defun es-variable (e)
  (some (lambda (c)
          (eql (char (symbol-name e) 0) c))
        '(#\X #\Y #\Z #\U #\V #\W)))
```

- Ejemplos:

```
> (es-variable 'x2)
T
> (es-variable 'a)
NIL
```

Unificación

- Sustitución $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}, n \geq 0$
  - Ejemplos: $\{X/a, Z/f(X,Y)\}$
    $\{X/a, Z/f(a,c)\}$
    $\epsilon = \{\}$
  - Representación en Lisp
    ```
    ((x . a) (z . (f x y)))
    ((x . a) (z . (f a c)))
    ()
    ```
- Dominio de una sustitución
  dominio$(\{X_1/t_1, \ldots, X_n/t_n\}) = \{X_1, \ldots, X_n\}$
  - Implementación: **`(dominio` σ`)`**
  - Ejemplo
    ```
    > (dominio '((x . a) (z . (f x y))))
    (X Z)
    ```


Unificación

- Aplicación de una sustitución a una expresión
  $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$

$X_i\sigma$	$= t_i$
$X\sigma$	$= X$, si $X \notin$ dominio(σ)
$f(s_1, \ldots, s_m)\sigma$	$= f(s_1\sigma, \ldots, s_m\sigma)$
$p(s_1, \ldots, s_m)\sigma$	$= p(s_1\sigma, \ldots, s_m\sigma)$
$(A \leftarrow B_1, \ldots, B_m)\sigma$	$= A\sigma \leftarrow B_1\sigma, \ldots, B_m\sigma$
$\{E_1, \ldots, E_n\}\sigma$	$= \{E_1\sigma, \ldots, E_n\sigma\}$

- Implementación: **`(aplica` E σ`)`**
- Ejemplo:
  ```
  > (aplica '(p (f x y) z) '((x . a) (z . (f x y))))
  (P (F A Y) (F X Y))
  ```

Unificación

- Composición de sustituciones

  $\sigma = \{X_1/t_1, \ldots, X_n/t_n\} \qquad \theta = \{Y_1/s_1, \ldots, Y_m/s_m\}$

  $\sigma\theta = \{X_i/t_1\theta : i \in [1,n], X_i \neq t_i\theta\} \cup$
  $\{Y_j/s_j : j \in [1,m], Y_j \notin \text{dominio}(\sigma)\}$

- Implementación: **`(composicion` σ_1 σ_2`)`**
- Ejemplos:

```
> (setf e '(p (f x y) z))
(P (F X Y) Z)
> (setf sigma '((x . y) (y . (f a z))))
((X . Y) (Y F A Z))
> (setf theta '((y . x) (z . (g x))))
((Y . X) (Z G X))
> (aplica (aplica e sigma) theta)
(P (F X (F A (G X))) (G X))
> (composicion sigma theta)
((Y F A (G X)) (Z G X))
> (aplica e (composicion sigma theta))
(P (F X (F A (G X))) (G X))
```


Unificación

- t_1 y t_2 son unificables si y sólo si existe una sustitución σ tal que $t_1\sigma = t_2\sigma$
- Una sustitución σ es un unificador de t_1 y t_2 si y sólo si $t_1\sigma = t_2\sigma$
- Ejemplos:
  - $f(X)$ y $g(Y,Z)$ no son unificables
  - Unificadores de $g(g(X))$ y $g(Y)$

Unificador	Instancia
$\{X/3, Y/g(3)\}$	$g(g(3))$
$\{X/f(U), Y/g(f(U))\}$	$g(g(f(U)))$
$\{X/Z, Y/g(Z)\}$	$g(g(Z))$
$\{Y/g(X)\}$	$g(g(X))$

Unificación

- Una sutitución σ_1 es más general que otra σ_2 si y sólo si existe otra sustitución θ tal que $\sigma_2 = \sigma_1\theta$
  - Comparación de unificadores de $g(g(X))$ y $g(Y)$

$\{X/3, Y/g(3)\}$	$\{Y/g(X)\}\{X/3\}$
$\{X/f(U), Y/g(f(U))\}$	$\{Y/g(X)\}\{X/f(U)\}$
$\{X/Z, Y/g(Z)\}$	$\{Y/g(X)\}\{X/Z\}$
$\{Y/g(X)\}$	$\{Y/g(X)\}\epsilon$

- σ es un unificador de máxima generalidad (UMG) de t_1 y t_2 si
  - σ es un unificador de t_1 y t_2
  - σ es más general que cualquier unificador de t_1 y t_2

Unificación

- Algoritmo de unificación: **`(unifica` E_1 E_2`)`**
  - Si $E_1 = E_2$, el proceso termina devolviendo $\{\}$
  - Si E_1 (E_2) es una variable
    - Si E_1 (E_2) aparece en E_2 (E_1), el proceso termina con fallo (**OccurCheck**)
    - En caso contrario, devolver la sustitución $\{E_1/E_2\}$ ($\{E_2/E_1\}$)
  - Si E_1 y E_2 son símbolos, el proceso termina con fallo (no son iguales)
  - En otro caso
    - Sea σ_1 el resultado de unificar el primer elemento de E_1 con el primer elemento de E_2
    - Si σ_1 no existe, el proceso termina con fallo
    - Sea σ_2 el resultado de unificar el resto de $E_1\sigma_1$ con el resto de $E_2\sigma_1$
    - Si σ_2 no existe, el proceso termina con fallo
    - En otro caso, el proceso termina devolviendo $\sigma_1\sigma_2$

Unificación: Ejemplos

- Unificación de $f(X, X)$ y $f(Y, a)$

```
(f X X)  (f Y a)
  (X X)    (Y a)  X/Y
    (Y)      (a)  Y/a
umg = X/YY/a = X/a, Y/a
```

- Unificación de $f(a, X)$ y $f(b, Y)$

```
(f a X)  (f b Y)
  (a X)    (b Y)  Fallo
```

- Unificación de $f(X, X)$ y $f(a, b)$

```
(f X X)  (f a b)
  (X X)    (a b)  X/a
    (a)      (b)  Fallo
```

- Unificación de $f(Y, X)$ y $f(X, g(Y))$

```
(f Y X)   (f X (g Y))
  (Y X)     (X (g Y))   Y/X
    (X)       ((g X))   Fallo
```


Unificación: Ejemplos

- Unificación de $f(X, X)$ y $f(Y, g(Y))$

```
(f X X)   (f Y (g Y))
  (X X)     (Y (g Y))   X/Y
    (Y)       ((g Y))   Fallo
```

- Unificación de $p(U, a, g(U))$ y $p(f(X, Y), X, Z)$

```
(p U a (g U))          (p (f X Y) X Z)
  (U a (g U))            ((f X Y) X Z)   U/(f X Y)
    (a (g (f X Y)))              (X Z)   X/a
      ((g (f a Y)))                (Z)   Z/(g (f a Y))
umg = U/(f a Y), X/a, Z/(g (f a Y))
```

- Unificación de $p(U, a, g(U))$ y $p(f(X, Y), X, Y)$

```
(p U a (g U))          (p (f X Y) X Y)
  (U a (g U))            ((f X Y) X Y)   U/(f X Y)
    (a (g (f X Y)))              (X Y)   X/a
      ((g (f a Y)))                (Y)   Fallo
```

Resolución SLD

- Lenguaje natural
  - Javier defiende a todos sus amigos
  - Todos los socios de un equipo son amigos de los partidaros de dicho equipo
  - Fernando es socio del betis
  - Javier es partidario del betis
- Programa

```
defiende(javier,X)        <- amigo_de(X,javier)
amigo_de(X,Y)             <- socio_de(X,Z), partidario(Y,Z)
socio_de(fernando,betis)  <-
partidario(javier,betis)  <-
```

- Conclusión: Javier defiende a Fernando
- Formalización: `<- defiende(javier,fernando)`

Resolución SLD

- Resolución SLD

```
<- defiende(javier,fernando)
   | defiende(javier,X1) <- amigo_de(X1,javier)
   | {X1/fernando}
<- amigo_de(fernando,javier)
   | amigo_de(X2,Y2) <- socio_de(X2,E2), partidario(Y2,E2)
   | {X2/fernando, Y2/javier}
<- socio_de(fernando,E2), partidario(javier,E2)
   | socio_de(fernando,betis) <-
   | {E2/betis}
<- partidario(javier,betis)
   | partidario(javier,betis) <-
   | {}
  <-
```

Resolución SLD: Cálculo de respuestas

- Programa

```
defiende(javier,X)        <- amigo_de(X,javier)
amigo_de(X,Y)             <- socio_de(X,Z), partidario(Y,Z)
socio_de(fernando,betis)  <-
partidario(javier,betis)  <-
```

- Objetivo: ¿A quién defiende Javier?
- Formalización: `<- defiende(javier,X)`
- Significado del objetivo: $\neg(\exists X)[defiende(javier, X)]$

Resolución SLD: Cálculo de respuestas

- Resolución SLD

```
<- defiende(javier,X0)
   | defiende(javier,X1) <- amigo_de(X1,javier)
   | {X1/X0}
<- amigo_de(X0,javier)
   | amigo_de(X2,Y2) <- socio_de(X2,Z2), partidario(Y2,Z2)
   | {X2/X0, Y2/javier}
<- socio_de(X0,Z2), partidario(javier,Z2)
   | socio_de(fernando,betis) <-
   | {X0/fernando, Z2/betis}
<- partidario(javier,betis)
   | partidario(javier,betis) <-
   | {}
  <-
```

- Respuesta: `{X/fernando}`

Resolución SLD: Cálculo de respuestas

- Programa

```
suma(0,X,X)         <-
suma(s(X),Y,s(Z)) <- suma(X,Y,Z)
```

- Objetivo: `<- suma(s(0),s(0),X)`
- Significado del objetivo: $\neg(\exists X)[suma(s(0), s(0), X)]$
- Resolución SLD

```
<- suma(s(0),s(0),X0)
    | suma(s(X1),Y1,s(Z1)) <- suma(X1,Y1,Z1)
    | {X1/0, Y1/s(0), X0/s(Z1)}
<- suma(0,s(0),Z1)
    | suma(0,X2,X2) <-
    | {X2/s(0), Z1/s(0)}
  <-
```

- Respuesta: `{X/s(s(0))}`

Resolución SLD: Cálculo de respuestas

- Programa

```
suma(0,X,X)         <-
suma(s(X),Y,s(Z)) <- suma(X,Y,Z)
```

- Objetivo: `<- suma(s(0),X,Y)`
- Significado del objetivo: $\neg(\exists X, Y)[suma(s(0), X, Y)]$
- Resolución SLD:

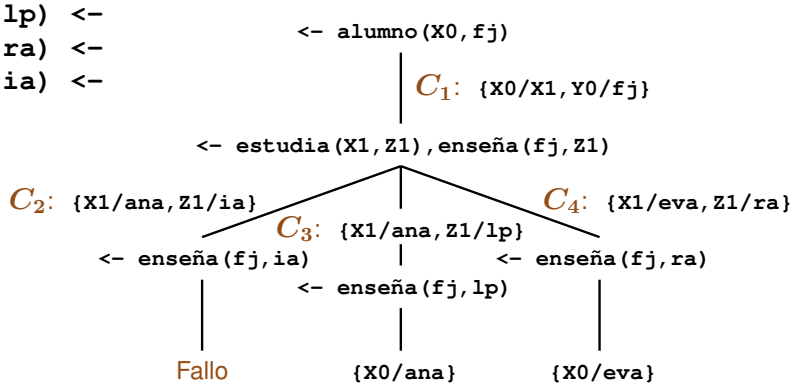
```
<- suma(s(0),X0,Y0)
    | suma(s(X1),Y1,s(Z1)) <- suma(X1,Y1,Z1)
    | {X1/0, Y1/X0, Y0/s(Z1)}
<- suma(0,X0,Z1))
    | suma(0,X2,X2) <-
    | {X2/X0, Z1/X0}
  <-
```

- Respuesta: `{Y/s(X)}`
- $P \models suma(s(0), X, s(X))$

Resolución SLD: Arbol SLD

- Programa

```
C1: alumno(A,P) <- estudia(A,C), enseña(P,C)
C2: estudia(ana,ia) <-
C3: estudia(ana,lp) <-
C4: estudia(eva,ra) <-
C5: enseña(fj,lp) <-
C6: enseña(fj,ra) <-
C7: enseña(jl,ia) <-
```

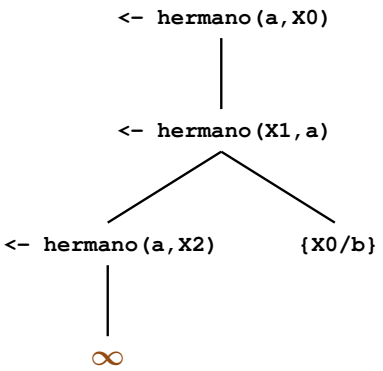


- Ramas de éxito y de fallo

Resolución SLD: Arbol SLD infinito

- Programa

```
hermano(X,Y) <- hermano(Y,X)
hermano(b,a)
```



Resolución SLD: Multidireccionalidad

- Programa

```
suma(0,X,X)
suma(s(X),Y,s(Z)) <- suma(X,Y,Z)
```

- Resta: `<- suma(s(0),X,s(s(s(0))))`
  - Respuesta: `{X/s(s(0))}`
- Descomposición en sumandos: `<- suma(X,Y,s(s(0)))`
- Respuestas múltiples:
  - R_1: `{X/0,Y/s(s(0))}`
  - R_2: `{X/s(0),Y/s(0)}`
  - R_3: `{X/s(s(0)),Y/0}`

Resolución SLD

- Adecuación y completitud básica de la resolución SLD
  - $Exito(P) = \{A \in BH(P) : P \cup \{\leftarrow A\} \vdash_{SLD} \square\}$
  - Adecuación y completitud:
    $Exito(P) = \{A \in BH(P) : P \models A\} = \text{MMH}(P)$
- Adecuación y completitud de la resolución SLD
- Sean P un programa lógico y $G = \leftarrow A_1, \ldots, A_n$ un objetivo
  - Adecuación: Si σ es una respuesta computada de $P \cup \{G\}$, entonces $P \models (A_1 \wedge \ldots \wedge A_n)\sigma$
  - Completitud: Si $P \models (A_1 \wedge \ldots \wedge A_n)\theta$, entonces existen σ, γ tales que σ es una respuesta computada de $P \cup \{G\}$ y $\theta = (\sigma\gamma)_{\restriction \text{Variables}(G)}$

Bibliografía

- Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).
  - Cap. 2: “Logic and resolution”
- Rich, E. y Knight, K. *Inteligencia artificial (segunda edición)* (McGraw–Hill Interamericana, 1994).
  - Cap. 5: “La lógica de predicados”
- Russell, S. y Norvig, P. *Inteligencia artificial (un enfoque moderno)* (Prentice–Hall, 1996).
  - Cap. 9: “La inferencia en la lógica de primer orden”
  - Cap. 10: “Sistemas de razonamiento lógico”
- Winston, P.R. *Inteligencia Artificial (3a. ed.)* (Addison–Wesley, 1994).
  - Cap. 13: “Lógica y prueba de resolución”

Capítulo 13

Implementación de Prolog

Lógica y Programación

Implementación de Prolog

J.-A. Alonso, F.-J. Martín-Mateos, J.-L. Ruiz-Reina

Dpto. 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

- Ejemplo de computación:

```
<- camino(x0,b)
    | (Alternativa 1)
    | camino(x1,z1) <- arco(x1,y1), camino(y1,z1)
    | {x1/x0, z1/b}
<- arco(x0,y1), camino(y1,b)
    | arco(a,b) <-
    | {x0/a, y1/b}
<- camino(b,b)
    | (Alternativa 2)
    | camino(x3,z3) <- arco(x3,y3), camino(y3,z3)
    | {x3/b, z3/b}
<- arco(b,y3), camino(y3,b)
    |
  Fallo y vuelta a (Alternativa 2)
```


Computación de respuestas

- Ejemplo de computación: (Alternativa 2)

```
<- camino(x0,b)
    | (Alternativa 1)
    | camino(x1,z1) <- arco(x1,y1), camino(y1,z1)
    | {x1/x0, z1/b}
<- arco(x0,y1), camino(y1,b)
    | arco(a,b) <-
    | {x0/a, y1/b}
<- camino(b,b)
    | (Alternativa 2)
    | camino(x3,x3) <-
    | {x3/b}
  <-
  Respuesta:  {x/a}
```

Computación de respuestas

- Ejemplo de computación: (Alternativa 1)

```
<- camino(x0,b)
   |  (Alternativa 1)
   |  camino(x1,z1) <-
   |  {x0/b, x1/b}
  <-
  Respuesta:  {x/b}
```


Computación de respuestas con entornos

- Ejemplo de computación:

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

Computación de respuestas con entornos

- Ejemplo de computación: (Alternativa 2)

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


Computación de respuestas con entornos

- Ejemplo de computación: (Alternativa 1)

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

Implementación de la resolución SLD

- 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 programas lógicos
  - Programa:
    ```
    p(x,z) <- q(x,y), p(y,z)
    p(x,x) <-
    q(a,b) <-
    ```
  - Representación:
    ```
    (setf *programa*
          '(((p x z) (q x y) (p y z))
            ((p x x))
            ((q a b))))
    ```

Implementación de la resolución SLD

- Expresiones anotadas

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

- Representación de objetivos
  - Pregunta: `<- p(a,x), q(x,b)`
  - Representación: `( (n (p a x)) (n (q x b)) )`
- Paso de SLD-resolución
  - Pregunta: `( (n (p a x)) (n (q x b)))`
  - Cláusula: `((p x z) (q x y) (p y z))`
  - Resolvente: `( (n+1 (q x y)) (n+1 (p y z)) (n (q x b)) )`

Implementación de la resolución SLD

- Nombre de un término anotado: **(nombre T_a)**
  - Ejemplo:

```
> (nombre '(1 (f x)))
(F X)
```

- Variables anotadas: **(es-variable-anotada T_a)**
  - Ejemplos:

```
> (es-variable-anotada '(3 y))
(Y Z U V W)
> (es-variable-anotada '(3 a))
NIL
```

- Símbolos anotados: **(es-simbolo-anotado T_a)**
  - Ejemplos:

```
> (es-simbolo-anotado '(3 f))
T
> (es-simbolo-anotado '(3 (f x)))
NIL
```

Implementación de la resolución SLD

- Manejo de expresiones anotadas
  - Implementación: **(primera-expresion L_a)**
  - Ejemplos:

```
> (primera-expresion '(1 (f (g x) (h y))))
(1 F)
> (primera-expresion '(1 ()))
(1 NIL)
```

  - Implementación: **(restantes-expresiones L_a)**
  - Ejemplos:

```
> (restantes-expresiones '(1 (f (g x) (h y))))
(1 ((G X) (H Y)))
> (restantes-expresiones '(1 ((h y))))
(1 NIL)
```


Implementación de la resolución SLD

- Representación de entornos
  - Entorno: $\{< n_i, x_i > / < m_i, t_i >: 1 \leq i \leq p\}$
  - Representación:
    (((n1 x1) . (m1 t1)) ... ((np xp) . (mp tp)))
- Término anotado asociado a una ligadura
  - Implementación: **(termino-anotado V_a ω)**
  - Ejemplo:

```
> (termino-anotado '(0 x) '(((0 x) . (1 a))))
(1 A)
```

- Variable anotada soportada por un entorno
  - Implementación: **(es-soportada V_a ω)**
  - Ejemplos:

```
> (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)
```

Implementación de la resolución SLD

- Añadir una ligadura a un entorno
  - Implementación: **(annade-ligadura** V_a T_a ω**)**
  - Ejemplos:

```
> (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))
```


Implementación de la resolución SLD

- Valor en un entorno
  - Implementación: **(valor** V_a ω**)**
  - Ejemplos:

```
> (defvar *entorno* '(((1 x) 2 b) ((0 x) 1 x)))
> (valor  ()    *entorno*)
NIL
> (valor '(0 y) *entorno*)
(0 Y)
> (valor '(1 x) *entorno*)
(2 B)
> (valor '(0 x) *entorno*)
(2 B)
> (valor '(0 (f x)) *entorno*)
(2 (f x))
```

Implementación de la resolución SLD

- Valor de una expresión anotada en un entorno
  - Implementación: **(aplica E_a ω)**
  - Ejemplos:

```
> (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))
```


Implementación de la resolución SLD

- Unificación de expresiones anotadas
  - Implementación: **(unifica $E1_a$ $E2_a$ ω)**
  - Ejemplos:

```
> (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

- 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 *programa*
      '(((P x z) (Q x y) (P y z))
        ((P x x))
        ((Q a b))))
> (prueba *programa* '((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 *programa* '((P x b) (P a a)))
| (SLD-RESOLUCION *programa*
|                 '((0 (P X B)) (0 (P A A)))
|                 '1
|                 'NIL)
|   (SLD-RESOLUCION *programa*
|                   '((1 (Q X Y)) (1 (P Y Z)) (0 (P A A)))
|                   '2
|                   '(((1 Z) 0 B) ((0 X) 1 X)))
|     (SLD-RESOLUCION *programa*
|                     '((1 (P Y Z)) (0 (P A A)))
|                     '3
|                     '(((1 Y) 2 B) ((1 X) 2 A) ((1 Z) 0 B) ((0 X) 1 X)))
|       (SLD-RESOLUCION *programa*
|       |               '((3 (Q X Y)) (3 (P Y Z)) (0 (P A A)))
|       |               '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)))
|       NIL
```

Implementación de la resolución SLD

- Traza de la prueba (cont)

```
(prueba *programa* '((P x b) (P a a)))
|
|       (SLD-RESOLUCION *programa*
|                       '((0 (P A A)))
|                       '4
|                       '(((3 X) 2 B) ((1 Y) 2 B) ((1 X) 2 A) ((1 Z) 0 B)
|                         ((0 X) 1 X)))
|         (SLD-RESOLUCION *programa*
|         |               '((4 (Q X Y)) (4 (P Y Z)))
|         |               '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)))
|         | (SLD-RESOLUCION *programa*
|         | |               '((4 (P Y Z)))
|         | |               '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)))
```


Implementación de la resolución SLD

- Traza de la prueba (cont)

```
(prueba *programa* '((P x b) (P a a)))
|         | |
|         | | (SLD-RESOLUCION *programa*
|         | | |               '((6 (Q X Y)) (6 (P Y Z)))
|         | | |               '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)))
|         | | NIL
|         | NIL
|         NIL
|         (SLD-RESOLUCION *programa*
|         |               '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

- Proceso de SLD–resolución:
  - Implementación: **(SLD-resolucion** P G N ω**)**
- Prueba de un objetivo:
  - Implementación: **(prueba** P G**)**
    - Nivel inicial: 0
    - Entorno inicial: $\{\}$
- Obtención de respuestas
  - Implementación: **(respuesta** P G**)**
- Variables iniciales
  - Implementación: **(variables** G**)**

Obtención de respuestas

- Aritmética natural

```
> (setf *suma*
    '(((suma 0 y y))
      ((suma (s x) y (s z)) (suma x y z))))

> (respuesta *suma* '((suma (s 0) (s 0) x)))
X = (S (S 0))
Si.
NIL
> (respuesta *suma* '((suma x (s 0) (s (s 0)))))
X = (S 0)
Si.
NIL
```

Obtención de respuestas

- Aritmética natural

```
> (setf *producto*
    '(((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* '((producto (s (s 0)) (s (s (s 0))) x)))
X = (S (S (S (S (S (S 0))))))
Si.
NIL
> (respuesta *producto*
             '((producto (s (s 0)) x (s (s (s (s (s (s 0))))))))))
X = (S (S (S 0)))
Si.
NIL
```


Obtención de respuestas

- Aritmética natural

```
> (setf *factorial*
    '(((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* '((factorial (s (s (s 0))) x)))
X = (S (S (S (S (S (S 0))))))
Si.
NIL
```

Obtención de respuestas

- Concatenación de listas

```
> (setf *append*
    '(((append nil x x))
      ((append (cons x y) z (cons x u)) (append y z u))))

> (respuesta *append* '((append (cons a (cons b nil)) (cons c nil) z)))
Z = (CONS A (CONS B (CONS C NIL)))
Si.
NIL
> (respuesta *append* '((append x (cons b nil) (cons a (cons b nil)))))
X = (CONS A NIL)
Si.
NIL
> (respuesta *append* '((append (cons x nil) y (cons a (cons b nil)))))
X = A
Y = (CONS B NIL)
Si.
NIL
```


Referencias

- Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).
  - Cap. 2: “Logic and resolution”
- Luger, G.F. y Stubblefield, W.A. *Artificial Intelligence (Structures and Strategies for Complex Problem Solving)* (Addison–Wesley, 1997).
  - Cap. 10.8 “Logic programming in Lisp”
- Sangal, R. *Programming Paradigms in Lisp* (McGraw–Hill, 1991).
  - Cap. 5 “Logic programming”
- Winston, P.H. y Horn, B.K. *Lisp (tercera edición)* (Addison–Wesley, 1991).
  - Cap. 27 “Encadenamiento regresivo y Prolog”