UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA

LENGUAJES FORMALES (75.14)

Trabajos Prácticos en Lisp

Demian Ferrerio 88443 epidemian@gmail.com

11 de julio de 2011

Índice

1. Introducción

Los trabajos prácticos se realizaron usando SBCL[1] y Clisp[2] como implementaciones de Common Lisp.

En las siguientes secciones se incluye el código fuente completo de cada ejercicio, una explicación de cómo se diseñó la solución y algunas pruebas para verificar su correcto funcionamiento.

2. GPS

El objetivo de este ejercicio es modelar calles e intereseccciones y diseñar un "GPS" que permita encontrar un camino para ir de un punto a otro.

2.1. Grafo Simple

Un grafo se modelará como una lista de elementos de la forma `(a (b1 ... bn)`, donde `a` y `b1 ... bn` son nodos distintos y `(a (b1 ... bn)` representa que existen aristas salientes de `a` y entrantes en `b1 ... bn`.

Por el momento, los nodos serán átomos.

```
; La representación del grafo no conexo:
;    a--c--f
;    |    / \    h--i
;    b---d---g    \
;     \     /      j
;      `-e-'
(defparameter *grafo-simple*
  '((a (b c)) (b (a d e)) (c (a f)) (d (b f g)) (e (b g)) (f (c d g))
    (g (d e f)) (h (i j)) (i (h)) (j (h))))
```

La variable global `*grafo-simple*` es entonces un grafo no conexo y no dirigido que luego se usará para probar la función GPS.

[1] http://www.sbcl.org/
[2] http://www.clisp.org/

Es importante notar que esta forma de modelar los grafos permite ahcer grafos dirigidos. Por ejemplo `'((a (b)) (b (c)))` es un grafo dirigido donde se puede ir de `a` hasta `c` pero no al revés. Esto se usará luego para modelar calles de mano única.

2.2. Encontrar un Camino

Para encontrar un camino será necesario comparar nodos. Para ello, conviene hacer una función que verifique si dos nodos son iguales.

```
(defun nodos-iguales (a b)
  (or (eql a b)))
```

Por ahora con derivar a `eql` alcanza, pues los nodos son simples átomos. Más adelante se redefinirá esta función para poder comparar esquinas de calles.

También será necesario encontrar, para un nodo dado, sus vecinos:

```
; Devuelve los nodos vecinos de un nodo en un grafo dado.
(defun vecinos (nodo grafo)
  (cadr (find nodo grafo :key 'car :test 'nodos-iguales)))
```

Nótese que se está usando la función `nodos-iguales` para comparar nodos.

Para probar que esta función anda correctamente, se puede ejecutar en el ambiente interactivo de Lisp:

```
> (vecinos 'f *grafo-simple*)
(C D G)
```

Con esto ya se puede hacer una primer implementación del GPS:

```
; Devuelve un camino que conecta los nodos inicio y fin en un grado dado, o nil
; en caso de no existir camino alguno.
(defun gps (inicio fin grafo &optional (trayectorias (list (list inicio))))
  (if (null trayectorias) nil
      (if (nodos-iguales (caar trayectorias) fin)
          (reverse (car trayectorias))
          (gps inicio fin grafo
               (append (expandir-trayectoria (car trayectorias) grafo)
                       (cdr trayectorias))))))

; Devuelve todas las posibles "expansiones" de una trayectoria dada agregando
; los vecinos del primer nodo que aún no formen parte de la misma.
(defun expandir-trayectoria (trayectoria grafo)
```

```
  (mapcar (lambda (vecino) (cons vecino trayectoria))
          (set-difference (vecinos (car trayectoria) grafo) trayectoria)))
```

Este algotritmo hace una búsqueda en profundidad de las trayectorias posibles hasta encontrar alguna que sea una camino que conecto los nodos `inicio` y `fin`. La función `expandir-trayectoria` devuelve todas las trayectorias posibles que resulten de tomar un nodo vecino del primer nodo que aún no forme parte de la trayectoria dada; puede devolver `nil` cuando no quedan más caminos por tomar para una trayectoria dada, y en ese caso es que se descartará esa trayectoria y se continuará expandiendo la siguiente en la función `gps` hasta cubrirlas todas.

Algunas puebras:

```
> (gps 'c 'c *grafo-simple*)
(C)
> (gps 'c 'e *grafo-simple*)
(C A B D F G E)
> (gps 'c 'i *grafo-simple*)
NIL
```

Se puede observar que el camino encontrado entre los nodos `c` y `e` no es el más corto posible, pero es una solución válida.

2.3. Camino Mínimo

Encontrar un camino mínimo a partir de las funciones ya definidas resulta sencillo:

```
; Devuelve un camino mínimo que conecta los nodos inicio y fin en un grado dado,
; o nil en caso de no existir camino alguno.
(defun gps-min (inicio fin grafo &optional (trayectorias (list (list inicio))))
  (if (null trayectorias) nil
      (or (reverse (find fin trayectorias :key 'car :test 'nodos-iguales))
          (gps-min inicio fin grafo
                   (mapcan (lambda (tr) (expandir-trayectoria tr grafo))
                           trayectorias)))))
```

Este algoritmo hace, en vez de una búsqueda en profundidad, una busqueda en anchura de las trayectorias posibles, es por eso que cuando encuentra una solución, ésta tiene que ser mínima. En cada llamada recursiva se checkea si alguna de las trayectorias es solución y en caso de que ninguna lo sea se vuelve a llamar recursivamente a `gps-min` pasándole todas las trayectorias expandidas.

Ejemplo de uso:

```
> (gps-min 'c 'e *grafo-simple*)
(C A B E)
```

Este camino sí es mínimo :)

2.4. Calles

Para hacer un grafo que represente un mapa de calles, los nodos deberán representar intersecciones entre calles. Para ello, pasarán de ser simples átomos a ser una lista de dos strings que serán los nombres de las calles que se intersectan en ese nodo. Por ejemplo `("Paseo Colon""Independencia")` es un nodo que representa la intersección entre las calles Paseo Colón e Independencia.

La definición de `nodos-iguales` cambiará acorde a la neuva definición de nodo:

```
; Verifica si dos nodos a y b son iguales.
(defun nodos-iguales (a b)
  (or (equalp a b)
      (and (listp a) (listp b) (equalp a (reverse b)))))
```

Esto es así porque se desea que `("Paseo Colon""Independencia")` y `("Independencia""Paseo Colon")` representen la misma intersección.

Para hacer las pruebas, se modelará parte de las calles aledañas a la Facultad de Ingeniería:

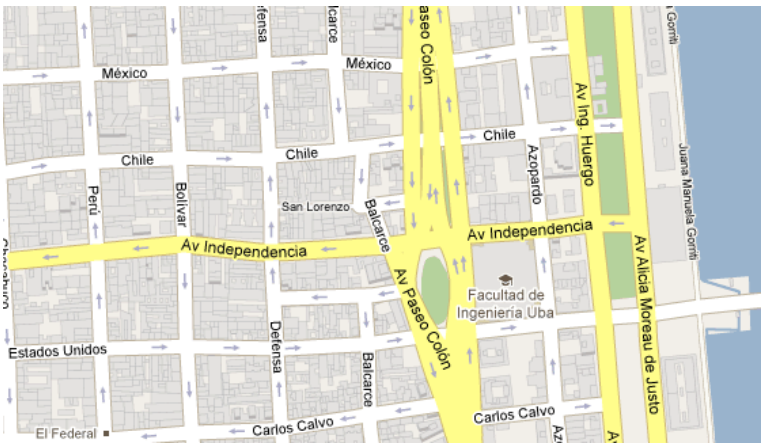

Figura 1: Mapa de la zona donde se ubica la Facultad de Ingeniería sacado de Google Maps.

Para hacer más sencilla la tarea de armar el grafo de calles, se numerarán las intersecciones que se desean modelar:

Figura 2: Enumeración de las intersecciones a modelar.

El grafo de estas calles se cargará en una variable global `*mapa-facu*`, que se define de la siguiente manera:

```
(defparameter *mapa-facu*
  (let* ((mx "Mexico") (ch "Chile") (in "Independencia") (eu "Estados Unidos")
         (pe "Peru") (bo "Bolivar") (de "Defensa") (ba "Balcarce")
         (pc "Paseo Colon") (az "Azopardo"))
    (crear-grafo (append (crear-calle mx (list pe bo de ba pc az))
                         (crear-calle ch (list pe bo de ba pc az))
                         (crear-calle in (list az pc de bo pe))
                         (crear-calle eu (list pe bo de pc az))
                         (crear-calle pe (list eu in ch mx))
                         (crear-calle bo (list mx ch in eu))
                         (crear-calle de (list eu in ch mx))
                         (crear-calle ba (list mx ch))
                         (crear-calle pc (list mx ch in eu))
                         (crear-calle pc (list eu in ch mx))
                         (crear-calle az (list eu in ch mx))))))
```

Las funciones `crear-grafo` y `crear-calle` pueden encontrarse el código fuente del programa (1). Lo que es importante saber es que el grafo creado respeta las manos de las calles, que son de mano única.

Gracias a que las calles son de mano única, los caminos mínimos no son tan directos:

```
> (gps-min '("Balcarce" "Mexico") '("Estados Unidos" "Bolivar") *mapa-facu*)
(("Balcarce" "Mexico") ("Mexico" "Paseo Colon") ("Paseo Colon" "Chile")
 ("Paseo Colon" "Independencia") ("Independencia" "Defensa")
 ("Independencia" "Bolivar") ("Bolivar" "Estados Unidos"))
```

2.5. Imprimir Camino

La función `imprimir-camino` es útil para hacer más legibles los caminos encontrados. Su definición puede encontrarse en el código fuente del programa (1). Por ahora, nos conformamos con una prueba:

```
> (imprimir-camino (gps-min '("Balcarce" "Mexico") '("Estados Unidos" "Bolivar")
*mapa-facu*))
Tomar "Mexico" hasta "Paseo Colon"
Tomar "Paseo Colon" hasta "Independencia"
Tomar "Independencia" hasta "Bolivar"
Tomar "Bolivar" hasta "Estados Unidos"
```

2.6. Código fuente

Listing 1: GPS

```
; Verifica si dos nodos a y b son iguales.
(defun nodos-iguales (a b)
  (or (equalp a b)
      (and (listp a) (listp b) (equalp a (reverse b)))))

; Devuelve los nodos vecinos de un nodo en un grafo dado.
(defun vecinos (nodo grafo)
  (cadr (find nodo grafo :key 'car :test 'nodos-iguales)))

; Devuelve todas las posibles "expansiones" de una trayectoria dada agregando
; los vecinos del primer nodo que aún no formen parte de la misma.
(defun expandir-trayectoria (trayectoria grafo)
  (mapcar (lambda (vecino) (cons vecino trayectoria))
          (set-difference (vecinos (car trayectoria) grafo) trayectoria)))

; Devuelve un camino que conecta los nodos inicio y fin en un grado dado, o nil
; en caso de no existir camino alguno.
(defun gps (inicio fin grafo &optional (trayectorias (list (list inicio))))
  (if (null trayectorias) nil
      (if (nodos-iguales (caar trayectorias) fin)
```

```
          (reverse (car trayectorias))
          (gps inicio fin grafo
               (append (expandir-trayectoria (car trayectorias) grafo)
                       (cdr trayectorias))))))

; Devuelve un camino mínimo que conecta los nodos inicio y fin en un grado dado,
; o nil en caso de no existir camino alguno.
(defun gps-min (inicio fin grafo &optional (trayectorias (list (list inicio))))
  (if (null trayectorias) nil
      (or (reverse (find fin trayectorias :key 'car :test 'nodos-iguales))
          (gps-min inicio fin grafo
                   (mapcan (lambda (tr) (expandir-trayectoria tr grafo))
                           trayectorias)))))

; La representación del grafo no conexo:
;    a--c--f
;    |    / \    h--i
;    b---d---g    \
;     \     /      j
;      `-e-'
(defparameter *grafo-simple*
  '((a (b c)) (b (a d e)) (c (a f)) (d (b f g)) (e (b g)) (f (c d g))
    (g (d e f)) (h (i j)) (i (h)) (j (h))))

; Devuelve el grafo que resulta de agregar una arista a un grafo dado. Una
; arista es una lista (n1 n2) y representa una conexión desde el nodo n1 al nodo
; n2. Ejemplo: (agregar-arista '(a c) '((a (b)) (b (c)))) -> ((a (c b)) (b (c)))
(defun agregar-arista (arista grafo)
  (if (member (cadr arista) (vecinos (car arista) grafo) :test 'nodos-iguales)
      grafo
      (cons (list (car arista)
                  (cons (cadr arista) (vecinos (car arista) grafo)))
            (remove-if (lambda (x) (nodos-iguales (car arista) (car x)))
                       grafo))))

; Crea un grafo a partir de una lista de aristas. Ejemplo:
; (crear-grafo '((a b) (b c))) -> ((b (c)) (a (b)))
(defun crear-grafo (aristas &optional (grafo-inicial nil))
  (if (null aristas) grafo-inicial
      (crear-grafo (cdr aristas)
                   (agregar-arista (car aristas) grafo-inicial))))

; Crea una lista de aristas a partir del nombre de una calle y sus
; intersecciones. Ejemplo (crear-calle "A" '("B" "C" "D")) ->
; ((("A" "B") ("A" "C")) (("A" "C") ("A" "D")))
(defun crear-calle (calle intersecciones)
```

```
    (if (< (length intersecciones) 2) nil
        (cons (list (list calle (car intersecciones))
                    (list calle (cadr intersecciones)))
              (crear-calle calle (cdr intersecciones)))))

(defparameter *mapa-facu*
  (let* ((mx "Mexico") (ch "Chile") (in "Independencia") (eu "Estados Unidos")
         (pe "Peru") (bo "Bolivar") (de "Defensa") (ba "Balcarce")
         (pc "Paseo Colon") (az "Azopardo"))
    (crear-grafo (append (crear-calle mx (list pe bo de ba pc az))
                         (crear-calle ch (list pe bo de ba pc az))
                         (crear-calle in (list az pc de bo pe))
                         (crear-calle eu (list pe bo de pc az))
                         (crear-calle pe (list eu in ch mx))
                         (crear-calle bo (list mx ch in eu))
                         (crear-calle de (list eu in ch mx))
                         (crear-calle ba (list mx ch))
                         (crear-calle pc (list mx ch in eu))
                         (crear-calle pc (list eu in ch mx))
                         (crear-calle az (list eu in ch mx))))))

; Dada una calle y una esquina (una lista con dos calles) devuelve la calle de
; la esquina que no es la calle dada. Ejemplo:
; (otra-calle-en-equina "A" ("B" "A")) -> "B"
(defun otra-calle (calle esquina)
  (find-if (lambda (x) (not (equalp calle x))) esquina))

; Imprime un camino en un formato legible.
(defun imprimir-camino (camino)
  (if (> (length camino) 1)
      (imprimir-camino-desde camino
                             (car (intersection (car camino) (cadr camino)
                                                :test 'equalp)))))

(defun imprimir-camino-desde (camino calle-anterior)
  (if camino
      (if (member calle-anterior (cadr camino) :test 'equalp)
          ; Sigue por la misma calle.
          (imprimir-camino-desde (cdr camino) calle-anterior)
          ; Agarra una calle nueva.
          (progn
            (format t "Tomar ~s hasta ~s~%" calle-anterior
                    (otra-calle calle-anterior (car camino)))
            (imprimir-camino-desde (cdr camino)
                                   (otra-calle calle-anterior (car camino)))))))
```

3. Intérprete TLC

3.1. Pruebas

La función principal del intérprete TLC es `evaluar`, que recibe dos argumentos: una expresión a evaluar y un ambiente. El ambiente es una lista de listas de la forma `nombre valor` donde `nombre` es el nombre de una variable y `valor` es el valor que tiene asociada. Así, por ejemlplo ...

```
> (evaluar '(cons a b) '((a 1) (b (2 3))))
(1 2 3)
```

...llama a `cons` con los valores `1` y `(2 3)`.

Algunas funciones de TLC Lisp se comportan distinto de las de Common Lisp. Por ejemplo, `nth` recibe primero la lista y luego el índice comenzando de 1:

```
> (evaluar '(nth (quote (a b c)) 2) nil)
B
```

En el ambiente se permiten definir funciones. Esto se hace asociando una función anónima a una variable. Por ejemplo:

```
(evaluar '(doble 6) '((doble (lambda (x) (* x 2)))))
12
```

Las funciones definidas en el ambiente pueden hacer referencia a variables definidas en él:

```
> (setq ambiente '((n 5)
                   (n-veces (lambda (x)
                              (* n x)))))
> (evaluar '(n-veces 10) ambiente)
50
```

Así, es posible definir funciones recursivas, como el factorial:

```
> (setq ambiente '((fact (lambda (n)
                           (if (eq n 0) 1
                               (* n (fact (- n 1))))))))
> (evaluar '(fact 5) ambiente)
120
```

3.2. Código fuente

Listing 2: Intérprete TLC Lisp

```
; Devuelve el valor de una variable en un ambiente dado.
(defun valor (nombre ambiente)
  (cadr (find nombre ambiente :key 'car)))

; Verifica si una función de TLC Lisp es directamente aplicable en Common Lisp
; sin tener que hacer ninguna traducción rara.
(defun es-funcion-aplicable (funcion)
  (member funcion '(car cdr cons list append + - * / < > eq atom null listp
                    numberp length)))

; Evalúa una expresión TLC Lisp en un ambiente dado. Por ejemplo:
; (evaluar '(cons x y) '((x 1) (y (2 3)))) -> (1 2 3)
(defun evaluar (expresion ambiente)
  (if (atom expresion)
      (if (numberp expresion)
          expresion
          (valor expresion ambiente))
      (cond
        ; Funciones que evalúan solo algunos de sus argumentos.
        ((eq (car expresion) 'quote) (cadr expresion))
        ((eq (car expresion) 'and) (and (evaluar (nth 1 expresion) ambiente)
                                        (evaluar (nth 2 expresion) ambiente)))
        ((eq (car expresion) 'or) (or (evaluar (nth 1 expresion) ambiente)
                                      (evaluar (nth 2 expresion) ambiente)))
        ((eq (car expresion) 'if) (if (evaluar (nth 1 expresion) ambiente)
                                      (evaluar (nth 2 expresion) ambiente)
                                      (evaluar (nth 3 expresion) ambiente)))
        ; Funciones que evalúan todos sus argumentos.
        (t (aplicar (car expresion)
                    (mapcar (lambda (x) (evaluar x ambiente)) (cdr expresion))
                    ambiente)))))

(defun aplicar (funcion argumentos ambiente)
  (if (atom funcion)
      (cond
        ; nth se comporta distinto en TLC.
        ((eq funcion 'nth) (nth (- (cadr argumentos) 1) (car argumentos)))
        ; Si es una función que se comporta igual en CL, la aplica directamente.
        ((es-funcion-aplicable funcion) (apply funcion argumentos))
        ; Es una función definida en el ambiente.
        (t (aplicar (valor funcion ambiente) argumentos ambiente)))
      ; Si no es un átomo es una función lambda. Evalúa su cuerpo en el ambiente
      ; que resulta de asociar sus parámetros a los argumentos mas el ambiente
```

```
      ; anterior.
      (evaluar (nth 2 funcion)
               (append (mapcar 'list (nth 1 funcion) argumentos) ambiente))))
```

Es interesante notar que la función `aplicar`, cuando evalúa una función definida en el ambiente, crea un nuevo ambiente que resulta del ambiente anterior mas los nombres de los parámetros de la función a evaluar y sus respectivos valores. Así es como las funciones definidas en el ambiente pueden hacer referencia a variables del ambiente además de sus parámetros.

4. Intérprete C

4.1. Pruebas

La función `valor` devuelve el valor de una expresión de pseudo-C para una memoria dada. La memoria, tal como en el intéprete de TLC, es una lista de pares `(nombre valor)`. Por ejemplo:

```
> (valor '(a + 5) '((a 2)))
7
```

Las expresiones que recibe esta función son expresiones tipo C, así que, por ejemplo, las funciones booleanas devuelven `0` o `1`:

```
> (valor '(a > 5) '((a 2)))
0
```

Las expresiones pueden ser muy complejas, y respetan el orden de precedencia de los operadores de C:

```
> (valor '(b / a == (b + 2) % a) '((a 5) (b 10)))
1
```

4.1.1. Factorial

Como caso de prueba interesante, se presenta un programa que lee un número de la entrada, calcula el factorial de ese número y lo imprime en la salida.

```
> (setq programa
'((int n)
```

```
  (int f)
  (main ((scanf n)
         (f = 1)
         (while (n > 0) (
                (f *= n)
                (n --)
         ))
         (printf f)
  ))
 ))
> (setq entrada '(5))
> (run programa entrada)
120
```

4.2. Código fuente

Listing 3: Intérprete de Pseudo-C

```
(defparameter *mapa-operadores*
  '((+= +) (-= -) (*= *) (/= /) (%= %) (++ +) (-- -)))

(defparameter *mapa-pesos*
  '((== 1) (!= 1) (< 2) (> 2) (<= 2) (>= 2) (+ 3) (- 3) (* 4) (/ 5) (% 6)))

; Retorna el operador binario asociado a un operador de asignación.
; Por ejemplo (buscar-operador '*=) -> *
(defun buscar-operador (op)
  (cadr (find op *mapa-operadores* :key 'car)))

; Verifica si un símbolo es un operador en C.
(defun es-operador (op)
  (member op *mapa-pesos* :key 'car))

; Retorna el peso de un operador en C. Mientras más grande su peso, mayor su
; precedencia.
(defun peso-operador (op)
  (cadr (find op *mapa-pesos* :key 'car)))

; Verifica si un símbolo es una variable en una memoria dada.
(defun es-variable (var mem)
  (member var mem :key 'car))

; Una condición de error para las variables no declaradas.
```

```
(define-condition variable-no-declarada (error)
  ((var :initarg :text :reader var)))

; Devuelve la memoria que resulta de asignar una variable a una memoria dada.
(defun asignar (var valor mem)
  (if (not (es-variable var mem))
      (error 'variable-no-declarada :var var))
  (cons (list var valor) (remove var mem :key 'car)))

; Agrega una nueva variable a una memoria dada.
(defun agregar-variable (var mem)
  (cons (list var) mem))

; Ejecuta un programa en pseudo-C
(defun run(prg ent &optional(mem nil))
  (if (null prg) nil
      (if (eq (caar prg) 'int)
          (run (cdr prg) ent (agregar-variable (nth 1 (car prg)) mem))
          (if (eq (caar prg) 'main)
              (ejec (nth 1 (car prg)) ent mem )
              'no-hay-main))))

(defun ejec (prg ent mem &optional (sal nil))
  (if (null prg) sal
      (cond
        ; printf: Ejecuta el resto del programa modificando la salida
        ((eq (caar prg) 'printf)
         (ejec (cdr prg) ent mem (append sal (list (valor (cdar prg) mem)))))
        ; scanf: Ejecuta el rest del programa sacando el primer valor de la
        ; entrada y asociandolo a una variable en memoria.
        ((eq (caar prg) 'scanf)
         (ejec (cdr prg) (cdr ent)
               (asignar (nth 1 (car prg)) (car ent) mem) sal))
        ; Asignaciones.
        ((es-variable (caar prg) mem)
         (if (eq (nth 1 (car prg)) '=)
             ; Ejecuta el resto del programa modificando la memoria.
             (ejec (cdr prg) ent
                   (asignar (caar prg) (valor (cddar prg) mem) mem) sal)
             ; Transforma las sentencias a += b en a = a + b
             ; Y a++ en a = a + 1
             (ejec (cons (list (caar prg) '= (caar prg)
                               (buscar-operador (nth 1 (car prg)))
                               (if (member (nth 1 (car prg)) '(+= -= *= /= %=))
                                   (cddar prg) 1))
                         (cdr prg))
```

```
                    ent mem sal)))
        ; Operadores pre-inrecmento/decremento. Los tranforma en post-inc/dec.
        ((member (caar prg) '(++ --))
         (ejec (cons (reverse (car prg))(cdr prg)) ent mem sal))
        ; If: agrega las sentencias correspondientes al comienzo del resto del
        ; programa.
        ((eq (caar prg) 'if)
         (ejec (append
                (if (eq (valor (nth 1 (car prg)) mem ) 0)
                    (if (eq (length (car prg)) 5) (nth 4 (car prg)) nil)
                    (nth 2 (car prg)))
                (cdr prg))
               ent mem sal))
        ; While: si la condición se cumple vuelve a ejecutar todo el programa
        ; más las sentencias en el while. Si no, ejecuta el resto del programa.
        ((eq (caar prg) 'while)
         (if (eq (valor (nth 1 (car prg)) mem) 0)
             (ejec (cdr prg) ent mem sal)
             (ejec (append (nth 2 (car prg)) prg) ent mem sal))))))

; Devuelve el valor de una variable en una memoria dada.
(defun valor-memoria (var mem)
  (if (not (es-variable var mem))
      (error 'variable-no-declarada :var var))
  (cadr (find var mem :key 'car)))

; Retorna una función booleana estilo C a partir de un operador booleano de
; Lisp.
(defun funcion-booleana (op)
  (lambda (x y) (if (funcall op x y) 1 0)))

; Devuelve una función válida de Lisp a partir de un operador binario de C.
(defun operador-a-funcion (op)
  (cond
    ((member op '(< > <= >=)) (funcion-booleana op))
    ((eq op '==) (funcion-booleana 'eq))
    ((eq op '!=) (funcion-booleana (lambda (x y) (not (eq x y)))))
    ((eq op '%) 'mod)
    (t op)))

; Devuelve los operandos que resultan de ejecutar el primer operador de
; operadores sobre los dos primeros operandos de operandos.
(defun operar (mem operadores operandos)
  (cons (apply (operador-a-funcion (car operadores))
               (list (valor (nth 1 operandos) mem)
               (valor (nth 0 operandos) mem)))
```

```
        (cddr operandos)))

; Devuelve el valor de una expresión para una memoria dada.
(defun valor (expresion mem &optional(operadores nil) (operandos nil))
  (if (atom expresion)
      (if (not (null expresion))
          (if (numberp expresion) expresion (valor-memoria expresion mem))
          (if (null operadores)
              (valor (car operandos) mem)
              (valor expresion mem (cdr operadores)
                     (operar mem operadores operandos))))
      (if (es-operador (car expresion))
          (if (null operadores)
              (valor (cdr expresion) mem (cons (car expresion) operadores)
                     operandos)
              (if (< (peso-operador (car operadores))
                     (peso-operador (car expresion)))
                  (valor (cdr expresion) mem
                         (cons (car expresion) operadores) operandos)
                  (valor (cdr expresion) mem
                         (cons (car expresion) (cdr operadores))
                         (operar mem operadores operandos))))
          (valor (cdr expresion) mem operadores
                 (cons (car expresion) operandos)))))
```

5. N Reinas

5.1. Tablero

Para modelar un tablero con n reinas se utiliza una lista de la forma $(r_1 r_2 \cdots r_n)$, donde r_i es el número de columna de la reina ubicada en la i-ésima fila.

5.2. Funciones en común

El ejercicio de las N Reinas se resolvió de varias formas con el fin de conseguir mejor performance. Las distintoas soluciones comparten algunas funciones básicas.

Listing 4: Funciones comunes para el problema de las N Reinas

```
; Genera una secuencia de enteros consecutivos de tamaño n comenzando por el
; valor inicio. Por ejemplo (iota 5 1) -> (1 2 3 4 5)
```

```
(defun iota (n &optional (inicio 0))
  (if (zerop n) nil
      (cons inicio (iota (- n 1) (+ inicio 1)))))

; Verifica si una lista contiene elementos repetidos.
(defun tiene-repetidos (l)
  (and (not (null l))
       (or (find (car l) (cdr l))
           (tiene-repetidos (cdr l)))))

; Verifica si un tablero es una solución del problema de N Reinas. Un tablero
; se representa mediante una lista de longitud n donde cada elemento es la
; columna de una reina y su índice es la fila.
(defun es-solucion (tablero)
  ; Un tablero es solución si no hay reinas en las mismas diagonales.
  (and (not (tiene-repetidos (mapcar '+ tablero (iota (length tablero)))))
       (not (tiene-repetidos (mapcar '- tablero (iota (length tablero)))))))
```

5.3. Primer solución

Se puede observar que los tableros válidos, al no poder tener dos reinas en la misma fila o columna, serán siempre permutaciones de la secuencia $(12\cdots n)$.

La primer solución lo que único que hace es calcular todas las permutaciones de $(12\cdots n)$ y filtrar las que sean solución.

Listing 5: Primer solución al problema de las N Reinas

```
; Calcula las permutaciones de una lista. La lista no debe tener elementos
; repetidos. Por ejemplo (permutaciones '(a b c))) ->
; ((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))
(defun permutaciones (l)
  (if (null l)
      '(())
      ; Por cada elemento de l, genero una lsita de todas las permutaciones de l
      ; con ese elemnto en la cabeza.
      (mapcan (lambda (elem)
                (mapcar (lambda (perm) (cons elem perm))
                        (permutaciones (remove elem l))))
              l)))

; Encuentra todas las soluciones al problema de N Reinas para un N dado.
(defun n-reinas (n)
  (remove-if-not 'es-solucion (permutaciones (iota n))))
```

Esta solución, si bien sencilla, es muy ineficiente, tanto en tiempo como en uso de memoria. En tiempo porque está evaluando todas las permutacioens posibles de un conjunto de n elementos, y eso es $n!$, lo cual, algortmicamente, es muy complejo. Y en memoria porque debe almacenar esas $n!$ permutaciones en memoria antes de filtrarlas.

5.4. Primer optimización

Una primer optimización es ir descartando las permutaciones que no sean solución a medida que se van calculando. Para hacer esto se recurre a la programación de alto orden: se le pasa la función `es-solucion` como parámetro a una función `filtrar-permutaciones` que filtra las permutaciones de una secuencia que cumplan con ese predicado.

Listing 6: Primer optimización: no se guardan todas las permutaciones en memoria

```
; Genera una secuencia de enteros consecutivos de tamaño n comenzando por el
; valor inicio. Por ejemplo (iota 5 1) -> (1 2 3 4 5)
(defun iota (n &optional (inicio 0))
  (if (zerop n) nil
      (cons inicio (iota (- n 1) (+ inicio 1)))))

; Verifica si una lista contiene elementos repetidos.
(defun tiene-repetidos (l)
  (and (not (null l))
       (or (find (car l) (cdr l))
           (tiene-repetidos (cdr l)))))

; Verifica si un tablero es una solución del problema de N Reinas. Un tablero
; se representa mediante una lista de longitud n donde cada elemento es la
; columna de una reina y su índice es la fila.
(defun es-solucion (tablero)
  ; Un tablero es solución si no hay reinas en las mismas diagonales.
  (and (not (tiene-repetidos (mapcar '+ tablero (iota (length tablero)))))
       (not (tiene-repetidos (mapcar '- tablero (iota (length tablero)))))))
```

Así, se previene utilizar desmesuradamente la memoria, pero aún así se siguen evaluando las $n!$ permutaciones, por lo cual el algoritmo sigue siendo muy malo en términos de tiempo.

5.5. Segunda optimización

Una última optimización sale de la observación de que, por ejemplo para el caso de $n = 8$, las permutaciones de la forma $(12x_3 \cdots x_8)$ nunca serán solución, no importa qué valores tomen $x_3 \cdots x_8$.

Con la misma idea de usar una función como predicado para evaluar las permutaciones a filtrar, se peude hacer una función, `filtrar-permutaciones-parciales` que descarte todas las permutaciones derivadas de una permutación parcial que no pase el predicado.

Afortunadamente, la función `es-solucion` es suficientemente versatil como para evaluar soluciones parciales:

```
> (es-solucion '(4 5))
NIL
```

Listing 7: Segunda optimización: no se prueban todas las permutaciones

```
; Genera una secuencia de enteros consecutivos de tamaño n comenzando por el
; valor inicio. Por ejemplo (iota 5 1) -> (1 2 3 4 5)
(defun iota (n &optional (inicio 0))
  (if (zerop n) nil
      (cons inicio (iota (- n 1) (+ inicio 1)))))

; Verifica si una lista contiene elementos repetidos.
(defun tiene-repetidos (l)
  (and (not (null l))
       (or (find (car l) (cdr l))
           (tiene-repetidos (cdr l)))))

; Verifica si un tablero es una solución del problema de N Reinas. Un tablero
; se representa mediante una lista de longitud n donde cada elemento es la
; columna de una reina y su índice es la fila.
(defun es-solucion (tablero)
  ; Un tablero es solución si no hay reinas en las mismas diagonales.
  (and (not (tiene-repetidos (mapcar '+ tablero (iota (length tablero)))))
       (not (tiene-repetidos (mapcar '- tablero (iota (length tablero)))))))
```

5.6. Pruebas

La solución para $n = 1$ es trivial:

```
> (n-reinas 1)
((0))
```

No existe solución para $n = 2$ y $n = 3$:

```
> (n-reinas 2)
NIL
> (n-reinas 3)
NIL
```

$n = 4$ tiene 2 soluciones:

```
> (n-reinas 4)
((2 0 3 1) (1 3 0 2))
```

Y $n = 8$ tiene 92 soluciones:

```
> (length (n-reinas 8))
92
```