

CLIPS - Code Snippets

Versión 0.8


Departament de Ciències de la Computació



FIB

Facultat d'Informàtica
de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

This work is licensed under the Creative Commons
Attribution-NonCommercial-ShareAlike License. 

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or
send a letter to:

Creative Commons,
559 Nathan Abbott Way, Stanford,
California 94305,
USA.

1. Programación con reglas	1
1.1. Introducción	1
1.2. El motor de inferencia	2
1.3. Variables y reglas	2
1.4. Los hechos y la memoria de trabajo	3
2. Code snippets	5
2.1. Introducción	5
2.2. Ejemplos	5
2.2.1. Calcular el factorial	5
2.2.2. Bubble sort	7
2.2.3. Máximo de unos valores	9
2.2.4. El viajante de comercio	11
2.3. El sistema experto en reparación de automóviles	21

CAPÍTULO 1

PROGRAMACIÓN CON REGLAS

1.1 Introducción

Usar lenguajes que se basan en la lógica como PROLOG o CLIPS necesita que se cambie la manera de pensar a la hora de programar. La diferencia fundamental es que con estos lenguajes los programas declaran lo que hay que hacer pero no especifican explícitamente la manera de obtenerlo, por ello se denominan lenguajes declarativos.

La razón por la que es así es porque debajo de ellos se encuentra un motor de inferencia que utiliza mecanismos de demostración automática para encontrar la solución que cumple las condiciones especificadas por el programa.

Esto significa que muchas veces para hacer que un programa encuentre una solución de manera eficiente debemos pensar en como funciona el motor de inferencia y que declaremos las condiciones del programa de acuerdo con ello. Además, este tipo de lenguajes incluyen habitualmente estructuras de control que permiten modificar la manera en la que el motor de inferencia hace la búsqueda de la solución para facilitar la eficiencia y dar algo de control al programador. Evidentemente la forma de pensar los programas puede depender de como funcione el motor de inferencia.

En lenguajes que se ejecutan con un motor de inferencia que hace razonamiento hacia atrás, la aproximación mas efectiva para desarrollar programas es pensar en que lo que se quiere hacer es una descomposición de problemas. Las reglas especificarán las condiciones que se han de cumplir para resolver cada subproblema. El motor de inferencia explorará todas las posibilidades y comprobará si existe alguna descomposición que cumpla con los datos del problema.

El tipo de preguntas que se resuelve con este tipo de programas siempre está bien establecida y suele seguir el esquema “¿Corresponde este conjunto de hechos a un problema del tipo X?”

En lenguajes que se ejecutan con un motor de inferencia que hace razonamiento hacia adelante la aproximación es diferente, principalmente porque son problemas en los que el objetivo final no esta especificado. Las reglas intentan obtener todas las consecuencias posibles hasta obtener una respuesta que satisfaga ciertas condiciones.

El tipo de preguntas que se resuelve con este tipo de programas es bastante mas abierto, dado que de hecho no se sabe a priori cual es la naturaleza de la respuesta. Estamos explorando qué posibles respuestas podemos obtener con los hechos introducidos esperando a que aparezca una que nos satisfaga.

Obviamente a la hora de crear programas de este tipo es importante tener un control mayor de

qué se explora y como se explora. Afortunadamente este tipo de razonamiento permite hacer esto.

1.2 El motor de inferencia

Es importante pensar que a diferencia de en los lenguajes imperativos el control de la ejecución no lo marcamos nosotros, sino el motor de inferencia. Por lo que, salvo algunas facilidades que nos permita el lenguaje que utilicemos, no sabemos exactamente como se va a ejecutar el programa.

De hecho será la estrategia de resolución de conflictos del motor la que tome esas decisiones. Podemos pensar que en este tipo de programación nosotros no decidimos la secuencia de instrucciones que seguirá el programa, sino que se decidirá dependiendo de las características (hechos) del problema.

Podemos pensar en este tipo de programación como en un puzzle en el que nosotros creamos las piezas (reglas) y estas son ensambladas de manera automática para crear el programa que resuelve nuestro problema.

Hemos de pensar también que, ser consciente de cual es la estrategia de resolución de conflictos del motor de inferencia puede influenciar en como hacemos el programa. Fundamentalmente podemos decidir como se describen las cosas en cada regla o el orden en que aparecen para que favorezcan una exploración mas eficiente de la solución.

Debemos pues desechar las ideas que tenemos de la programación imperativa, nosotros no controlamos nada (o pocas cosas).

Eso que en principio podría parecer malo, de hecho permite expresar de manera mas eficiente y sucinta programas que sería bastante complejos para la programación imperativa. Estamos renunciando al control para obtener mas nivel de abstracción, para preocuparnos mas del *qué* y menos de *cómo*.

Obviamente, no todos los problemas se pueden expresar fácilmente en el paradigma declarativo, digamos que cada paradigma de programación tiene sus puntos fuertes y puntos débiles.

1.3 Variables y reglas

Algo que choca bastante cuando se empieza con la programación declarativa es que las variables tienen un comportamiento totalmente diferente. Este comportamiento varía dependiendo de si el método de razonamiento es hacia adelante o hacia atrás.

Primero de todo, todas las variables son locales a cada regla, por lo que no son visibles desde ninguna otra regla. Por lo tanto si queremos pasar algún tipo de resultado entre las reglas se deben utilizar mecanismos diferentes de paso de información.

En el caso del razonamiento hacia adelante esta comunicación se realiza básicamente a partir de hechos. Cada regla que quiera pasar algún tipo de información a otras reglas deberá generar algún tipo de hecho que permita ser recogido por otra regla.

En el caso de razonamiento hacia atrás se pueden utilizar también hechos para la comunicación entre reglas, pero el mecanismo mas habitual es la unificación de las variables de las condiciones de las reglas. Al ser cada condición un subobjetivo, las reglas que permitan cumplir ese subobjetivo unificarán sus variables con la instancia del subobjetivo correspondiente, obteniendo así sus valores.

Otro elemento diferencial es la asignación, se puede decir que no existe como tal. Esto es debido a que al ser cada regla independiente y al comunicarse a través de otros medios, ésta no es necesaria.

No obstante existen algunos lenguajes de reglas que se alejan un poco del paradigma declarativo y permiten la asignación por lo general de manera local en cada regla¹.

1.4 Los hechos y la memoria de trabajo

Como la comunicación entre las diferentes partes del programa se realiza a partir de hechos, la memoria de trabajo jugará un papel fundamental dentro de la programación en este tipo de lenguajes, sobre todo en los lenguajes que utilizan razonamiento hacia adelante.

Los hechos se podrán utilizar para comunicar información entre reglas, ya que al crear nuevos hechos se podrán instanciar reglas que recuperarán la información almacenada en esos hechos. Obviamente la generación de estos hechos permitirá además obtener el resultado del programa.

Otra utilidad esencial de los hechos es establecer mecanismos de control que permitan modificar como hace la exploración el motor de inferencia. Se pueden crear hechos que permitan activar o desactivar la ejecución de las reglas y así imponer cierto orden de ejecución.

¹Otros permiten asignaciones a variables globales, pero obviamente eso es una herejía independientemente del paradigma de programación usado.

2.1 Introducción

Un *code snippet* es una porción de código que ilustra como resolver un problema específico. Es habitual encontrar colecciones de ejemplos en muchos lenguajes, principalmente imperativos, aunque también existen ejemplos de lenguajes declarativos como PROLOG.

Desgraciadamente no hay mucho en lo que respecta a ejemplos de lenguajes que utilizan razonamiento hacia adelante. Este capítulo presenta ejemplos de programación utilizando reglas hacia adelante y técnicas y esquemas de programación que pueden ser útiles a la hora de desarrollar aplicaciones que las utilicen.

Dado que el lenguaje CLIPS permite utilizar además del lenguaje de reglas un lenguaje pseudo-funcional, hay cosas que se pueden programar directamente usando un estilo imperativo con este lenguaje obviando el paradigma declarativo. Obviamente, es mejor usarlo solamente cuando sea imprescindible.

El uso correcto de un lenguaje de reglas requiere cierto esfuerzo y el cambiar la manera de pensar la programación. No obstante este esfuerzo se ve recompensado con el acceso a un cambio de visión en la forma de programar, que incluso ayuda a pensar mejor al programar de manera imperativa.



Antes de empezar con los ejemplos deberíais repasar el funcionamiento de un motor de inferencia y la estrategia de resolución por encadenamiento hacia adelante.

2.2 Ejemplos

2.2.1 Calcular el factorial

Calcular el factorial es bastante sencillo en un lenguaje imperativo. Plantearlo como reglas hacia adelante requiere pensar un poco más. Primero hemos de tener en cuenta que las reglas son elementos independientes que necesitan hechos para poder ser ejecutadas. Por lo tanto necesitaremos un hecho que defina el ser el factorial de un numero.

Llamemos a este hecho **fact** y pensemos en él como una relación que liga un número con su factorial. Podemos decir que por ejemplo (**fact** 3 6) esta representando que el factorial de 3 es 6.



La manera mas sencilla de representar hechos en CLIPS es mediante **unordered facts**, con ellos establecemos relaciones entre elementos de cualquier tipo, la única restricción es que el primer elemento del hecho sea un **símbolo**.

Dado que estamos usando reglas hacia adelante lo que podrá hacer nuestro programa será calcular el factorial de un número dado otro. Obviamente la regla lo que hará será establecer la recurrencia entre un factorial y el del número siguiente.

```

1 (defrule factorial
2   (fact ?x ?y)
3   =>
4   (assert (fact (+ ?x 1) (* ?y (+ ?x 1))))
5 )

```

La regla simplemente comprueba si hay un hecho **fact** y crea como consecuencia un nuevo hecho que relaciona el siguiente valor con su factorial aplicando la recurrencia. Esta regla por si sola no hará nada si no creamos algún hecho que permita ejecutarla alguna vez. La opción obvia es crear el hecho que define el primer factorial:

```

1 (deffacts hechos
2   (fact 1 1)
3 )

```

Ahora la regla podrá ejecutarse, pero si ponemos en marcha el motor de inferencia se calcularán todos los factoriales que existen y nunca se acabará la ejecución. Lo que echamos de menos de nuestro programa imperativo es decirle que factorial queremos calcular.

A una regla no le podemos pasar parámetros, por lo que debemos introducir hechos que nos controlen cuando debemos acabar de calcular. Podemos introducir por ejemplo el hecho **limite**, que nos diga cual es el valor del que queremos hacer el calculo (por ejemplo 6):

```

1 (deffacts hechos
2   (fact 1 1)
3   (limite 6)
4 )

```

También debemos modificar la regla de manera que deje de ejecutarse cuando llegue a ese limite:

```

1 (defrule factorial
2   (limite ?l)
3   (fact ?x&:(< ?x ?l) ?y)
4   =>
5   (assert (fact (+ ?x 1) (* ?y (+ ?x 1))))
6 )

```

Básicamente lo que hacemos es recuperar el hecho que nos guarda el límite y comprobamos en la condición del factorial que el valor del número sea menor que el límite.

Un fallo que tiene esta regla es que nos deja todos los cálculos intermedios como hechos en la memoria de trabajo. Desde un punto de vista imperativo no tiene mucho sentido, por lo que podríamos eliminarlos una vez los hayamos utilizado:

```

1 (defrule factorial
2   (limite ?l)
3   ?f <-(fact ?x&:(< ?x ?l) ?y)
4   =>
5   (retract ?f)
6   (assert (fact (+ ?x 1) (* ?y (+ ?x 1))))
7 )

```

También podríamos no eliminar los hechos y aprovechar que los tenemos para no tener que calcular siempre desde el principio. Esto requeriría ciertas modificaciones en las reglas que se dejan para el lector.

Podemos ver que la regla de hecho está reproduciendo el comportamiento de una estructura iterativa en la que los hechos son los que controlan ese comportamiento a la vez que realizan el cálculo correspondiente. La iteración se obtiene por como hace la exploración el motor de inferencia.



Una mejora podría ser añadir reglas que permitieran preguntar el factorial que se quiere calcular e imprimieran la solución al final del cálculo. ¿Como se podría solucionar?

2.2.2 Bubble sort

El algoritmo del Bubble sort es uno de los algoritmos mas simples de ordenación de una estructura lineal (vector, lista, ...). Podemos crear reglas que permitan la ordenación de una estructura que nos simule por ejemplo un vector. En CLIPS no tenemos vectores, por lo que deberemos simularlos utilizando hechos. Para ello en lugar de **unordered facts** como hemos utilizado en el ejemplo anterior utilizaremos un **deftemplate**, para utilizar elementos diferentes de CLIPS, pero se puede hacer perfectamente con los primeros.



Los **deftemplates** permiten definir la estructura de los hechos, por lo que nos permiten definir hechos complejos y acceder fácilmente a las características que los representan. La sintaxis de CLIPS nos permite definir esas características de manera bastante libre. De hecho CLIPS es un lenguaje débilmente tipado, por lo que podemos definir características sin indicar su tipo. Esto es una ventaja a veces, por que nos permite incluir implícitamente genericidad, pero exige una mayor disciplina a la hora de programar.

Definiremos el template que nos servirá de posiciones del vector:

```

1 (deftemplate pos
2   (slot index (type INTEGER))
3   (slot value (type INTEGER))
4 )

```

Y con el podemos declarar un conjunto de hechos que nos hagan de vector:

```

5 (def facts vector
6   (pos (index 1) (value 15))
7   (pos (index 2) (value 7))
8   (pos (index 3) (value 4))
9   (pos (index 4) (value 2))
10 )

```

Obviamente podríamos haber escogido que las posiciones no fueran consecutivas, pero es mas claro así. Cada hecho es un elemento individual, por lo que no existe tal vector en la memoria de trabajo, es la interpretación que hacemos nosotros.

Ahora solo nos hace falta el programa que nos haga la ordenación. El bubble sort lo único que hace es intercambiar dos posiciones consecutivas si no están en orden, y eso es precisamente lo que tiene que decir la regla:

```

11 (defrule ordena-bubble
12   ?f1 <- (pos (index ?p1) (value ?v1))
13   ?f2 <- (pos (index ?p2&:(= ?p2 (+ ?p1 1))) (value ?v2&:(< ?v2 ?v1)))
14   =>
15   (modify ?f1 (value ?v2))
16   (modify ?f2 (value ?v1))
17 )

```

La condición de la regla establece que debe haber dos posiciones consecutivas fuera de orden, en ese caso lo que se hace es intercambiar los valores.

Aquí la relación con la versión imperativa no es tan obvia, aparentemente no hay ningún bucle que recorra las diferentes posiciones. De hecho todo el trabajo lo realiza el motor de inferencia, que realizará los intercambios que hacen los dos bucles de la versión imperativa. Hay que observar que no hay ningún control sobre en qué orden se realizarán los intercambios y solo se accederá a las posiciones que necesiten ser intercambiadas. En la versión imperativa se pueden recorrer posiciones sin que haya ningún intercambio.



Este tipo de programación, de hecho, nos permite pensar en una ejecución en paralelo de los algoritmos, donde si no estuviéramos restringidos por la ejecución secuencial del motor de inferencia podríamos hacer varios pasos a la vez ejecutando varias reglas al mismo tiempo.

Para completar el programa podemos hacer una regla que nos imprima en orden los valores de las posiciones de nuestro vector virtual. Parecería difícil hacer que se impriman las cosas en orden, pero todo es cuestión de expresar las condiciones de manera adecuada. La regla que hace esto es la siguiente:

```

18 (defrule imprime-res
19   (declare (salience -10))
20   ?f1 <- (pos (index ?p1) (value ?v))
21   (forall (pos (index ?p2)) (test (<= ?p1 ?p2)))
22   =>

```

```

23  (printout t ?v crlf)
24  (retract ?f1)
25  )

```

La condición simplemente dice que queremos una posición para la que todas las posiciones sean mayores o iguales que ella. La ejecución de la regla la controla la existencia de los hechos que representan el vector. Al eliminar la posición de la memoria de trabajo quedará la siguiente que queremos imprimir, hasta que no quede ninguna. El declarar la **salience** de la regla sirve para que esta regla se ejecute una vez haya de acabado de usarse la regla que hace la ordenación.

La segunda condición se podría haber escrito también como:

```
(not (pos (index ?p2:(< ?p2 ?p1))))
```

Es este caso estamos diciendo que no debe haber un hecho que tenga un índice menor que el que instancia la primera condición.



En este programa hemos usado las condiciones de las reglas para expresar lo que deseamos obtener. Este es un buen momento para que repaséis como se expresan las condiciones en CLIPS y penséis ejemplos de como se pueden utilizar estas para programar otros algoritmos de ordenación.



CLIPS tiene como estructura primitiva las listas. Estas se pueden indexar como si fueran vectores, por lo que se puede programar este algoritmo de ordenación igual que se hace de manera imperativa.

Un buen ejercicio para aprender como usar el lenguaje funcional de CLIPS sería programar una función que hiciera la ordenación de una lista de elementos.

2.2.3 Máximo de unos valores

Calcular el máximo puede parecer algo complicado de hacer utilizando reglas. Si pensamos en la versión imperativa debemos tener los datos en una estructura y recorrerla quedándonos durante el recorrido con el máximo valor que nos encontremos.

Podemos reproducir perfectamente este comportamiento utilizando reglas. Para la estructura solo necesitamos un conjunto de hechos que nos almacenen los valores de los que queremos calcular el máximo, utilizar la misma relación en cada hecho nos servirá para tener de manera virtual esa estructura que tenemos en la versión imperativa. Si queremos reproducir el comportamiento imperativo podemos tener un hecho adicional que nos vaya guardando el máximo valor que nos vayamos encontrando. El siguiente conjunto de reglas nos podría calcular el máximo de un conjunto de valores:

```

1  (defacts maximo
2    (val 12)
3    (val 33)
4    (val 42)
5    (val 56)
6    (val 64)
7    (val 72)

```

```

8  )
9
10 (defrule init "Inicializa el calculo del maximo"
11   (not (max ?))
12   =>
13   (assert (max 0))
14 )
15
16 (defrule calcula-max
17   (val ?x)
18   ?h <- (max ?y&:(> ?x ?y))
19   =>
20   (retract ?h)
21   (assert (max ?x))
22 )
23
24 (defrule print-max
25   (declare (salience -10))
26   (max ?x)
27   =>
28   (printout t ?x crlf)
29 )

```

En este caso la relación `val` es la que nos agrupa todos los valores y el hecho `max` el que nos guarda el valor máximo. La primera regla es la que inicia el calculo creando el hecho `max` con el mínimo valor, que es lo que hacemos en el algoritmo imperativo y la regla `calcula-max` es la que va comparando los valores con el valor máximo actual. La última regla simplemente imprime el valor una vez el valor máximo ya ha sido calculado. Eso lo sabemos porque ya no se puede ejecutar la regla `calcula-max`.

Aunque aparentemente el algoritmo que estamos usando es el mismo que en la versión imperativa, el comportamiento es bastante diferente. En primer lugar en el algoritmo imperativo somos nosotros los que decidimos en que orden se recorren los valores, en este algoritmo es el motor de inferencia el que lo decide, por lo que dependiendo de la estrategia de resolución de conflictos las comparaciones que realmente se hacen pueden variar notablemente.

Si por ejemplo los hechos que instancian primero la regla `calcula-max` son `(val 72)` y `(max 0)` ya no se harán mas ejecuciones de la regla ya que ese será el valor máximo. Si se fija uno en lo que dicen las condiciones de la regla lo que se pide es que haya un hecho `val` y que su valor sea mayor que el del hecho `max`. Una vez `max` tenga el máximo valor posible ya se habrá acabado y no se buscará mas.

Eso quiere decir que si usamos inteligentemente las condiciones de las reglas y cómo explora el motor de inferencia, podemos ser mas eficientes que programando de manera imperativa.



Un ejercicio interesante sería implementar las reglas que reprodujeran el comportamiento exacto del algoritmo imperativo de manera que se recorrieran secuencialmente todas las posiciones para calcular el máximo. La manera en que se implementó el algoritmo del bubble sort puede daros una pista.

Si queremos ser un poco mas inteligentes podemos olvidarnos de la versión imperativa del algoritmo y declarar en una regla lo que estamos buscando, que es simplemente un hecho que tenga un valor que sea mayor o igual que todos los otros posibles. La condición de igual viene de que evidentemente

el hecho con el valor máximo participa de la comparación, por lo que entre las posibles comparaciones está la que hace que el máximo se compare consigo mismo. Esto lo podemos escribir en una sola regla:

```

1 (defrule calcula-max
2   (val ?x)
3   (forall (val ?y) (test (>= ?x ?y)))
4   =>
5   (printout t ?x crlf)
6 )

```

Puede parecer milagroso, evidentemente no lo es. Aquí el esfuerzo lo hace el sistema de unificación del motor de inferencia. Debe hacer todas las comparaciones posibles y quedarse con la única que cumple las condiciones. Esta forma de describir como calcular el máximo es un ejemplo de la filosofía declarativa, el indicar que queremos, pero sin preocuparnos de como se calcula, ese es trabajo del motor de inferencia.



Este esquema permite resolver algoritmos en los que buscamos un elemento que cumple ciertas condiciones de entre un conjunto. En programación declarativa no es necesario hacer un algoritmo que haga la búsqueda, solo hay que hacer la consulta a la base de hechos de manera adecuada y el motor de inferencia nos dará el elemento que buscamos. Podemos pensar que la base de hechos nos hace el papel de una base de datos y el motor de inferencia es el que ejecuta las consultas.

2.2.4 El viajante de comercio

Como ya debéis conocer, el problema del viajante de comercio consiste en encontrar un camino que recorra un conjunto de ciudades, éste debe empezar y acabar en la misma ciudad y no debe recorrer una ciudad mas de una vez.

Este problema se puede resolver de manera aproximada utilizando el algoritmo de hill-climbing. Para hacer la exploración se pueden utilizar diferentes operadores, nosotros solo vamos a usar el intercambio de dos ciudades del recorrido. De esta manera habrá $\frac{N*(N-1)}{2}$ intercambios posibles en cada iteración.

Vamos a usar algo mas complicado para representar los elementos del problema. En este caso usaremos dos deftemplates para definir la matriz de distancias y la solución.

```

1 (deftemplate solucion
2   (multislot camino)
3   (slot coste (type INTEGER))
4   (slot desc)
5 )
6
7 (deftemplate mat-dist
8   (multislot dist)
9 )

```

En el caso de `solucion` tenemos tres slots, el camino de la solución, su coste, y un campo que utilizaremos para controlar la búsqueda, que indica si el hecho corresponde a la solución actual o a sus posibles descendientes. En el caso de `mat-dist` almacenamos la matriz de distancias entre ciudades. Esta está representada como una lista que corresponde a la parte triangular inferior (o superior, da igual) de la matriz de distancias.



Es habitual en la programación con reglas que se incluyan elementos en la representación que permitan controlar como se ejecutan las reglas. En este caso en la representación de solución el slot `desc` se utilizará para distinguir entre los pasos que hacen la generación de los sucesores y los que hacen la selección del mejor sucesor.

Tenemos también dos funciones, una para acceder a la matriz de distancias y otra para calcular el coste de un camino.

```

10   ;;; Distancia entre dos ciudades
11   (deffunction calc-dist (?i ?j ?nciu ?m)
12     (if (< ?i ?j)
13       then (bind ?pm ?i) (bind ?pn ?j)
14       else (bind ?pm ?j) (bind ?pn ?i)
15     )
16     (bind ?off 0)
17     (loop-for-count (?i 1 (- ?pm 1))
18       do
19         (bind ?off (+ ?off (- ?nciu ?i)))
20       )
21     (nth$ (+ ?off (- ?pn ?pm)) ?m)
22   )
23
24   ;;; Coste de una solucion
25   (deffunction calc-coste (?s ?m)
26     (bind ?c 0)
27     (loop-for-count (?i 1 (- (length$ ?s) 1))
28       do
29         (bind ?c (+ ?c (calc-dist (nth$ ?i ?s) (nth$ (+ ?i 1) ?s) (length$ ?s) ?m)))
30       )
31     (bind ?c (+ ?c (calc-dist (nth$ 1 ?s) (nth$ (length$ ?s) ?s) (length$ ?s) ?m)))
32   )

```

La primera transforma los índices que recibe en la posición adecuada en la lista que representa la matriz de distancias, la segunda recorre la lista que representa el camino sumando las distancias.



En CLIPS las listas son un tipo primitivo y dispone de un conjunto de operadores para poder manipularlas. Habitualmente es necesitaréis utilizar estas estructuras en vuestros programas, por lo que sería un buen momento para mirar como se utilizan en el manual de CLIPS.

Para indicar para cuantas ciudades queremos calcular el problema tendremos un hecho:

```

33   (defacts TSP
34     (num-ciudades 10)
35   )

```


Para generar el programa que nos hace la exploración del hill climbing primero deberemos analizar un poco que es lo que se debe hacer. Tendremos que explorar soluciones de manera iterativa partiendo de una solución inicial. Para cada iteración deberemos generar todos los posibles intercambios y quedarnos con el mejor, acabaremos cuando a partir de la solución actual ya no haya ningún descendiente que la mejore.

Para calcular todos los intercambios posibles podemos utilizar el motor de inferencia para que nos haga él mismo los cálculos. Para ello solo necesitaremos un hecho auxiliar que nos va a controlar la generación de combinaciones. A este hecho lo llamaremos `pos` tendremos tantos hechos como posiciones y cada uno de ellos tendrá uno de los valores posibles. Para generar todos los pares solo necesitamos una condición del estilo `(pos ?i) (pos ?j)`. Si tenemos los hechos `pos` en la memoria de trabajo esto se instanciará a todas las posibles combinaciones de parejas. Como no queremos hacer mas trabajo de la cuenta, podemos añadir la restricción `(pos ?i) (pos ?j&:(> ?j ?i))`, de manera que solo se instancien las parejas que nos interesan.



En muchos programas es habitual necesitar generar todas las combinaciones de un conjunto de elementos. En programación imperativa nosotros hemos de escribir el programa que haga la generación. En programación declarativa podemos aprovechar el mecanismo de unificación del motor de inferencia para escribir reglas que hagan la generación de combinaciones. Un buen ejercicio sería pensar en otros problemas en los que sea necesario generar combinaciones de elementos y escribir las reglas que las generan.

Para pensar mejor en el algoritmo podemos estructurarlo en los siguientes pasos:

1. Inicializar la búsqueda creando una solución inicial
2. Generar todos los descendientes mejores a partir de intercambios
3. Escoger el mejor y actualizar la solución o acabar si no hay nuevas soluciones

Hay que darse cuenta de que cada vez que se actualiza la solución actual la regla que explore todos los intercambios posibles se podrá instanciar de nuevo al cambiar el hecho de la solución. Esto obviamente nos ahorra tener que pensar en función de iteraciones, lo hace todo el motor de inferencia.

Primero haremos una solución simple, solamente pensando en el esquema que hemos indicado y después pensaremos un poco en como hace la exploración el motor de inferencia para hacer las cosas un poco mas *limpias*. Comenzaremos con la regla que inicializa la búsqueda:

```

36 (defrule init
37   (num-ciudades ?x)
38   =>
39   ; Matriz de distancias
40   (bind ?m (create$))
41   (loop-for-count (?i 1 (/ (* ?x (- ?x 1)) 2))
42     do
43       (bind ?m (insert$ ?m ?i (+ (mod (random) 50) 1)))
44   )
45   (assert (mat-dist (dist ?m)))
46   ; Hechos de control para los intercambios
47   (loop-for-count (?i 1 ?x)

```

```

48   do
49     (assert (pos ?i))
50   )
51   (bind ?s (create$))
52   ; Solucion inicial (ciudades ordenadas)
53   (loop-for-count (?i 1 ?x)
54     do
55       (bind ?s (insert$ ?s ?i ?i))
56     )
57   (assert (solucion (camino ?s) (coste (calc-coste ?s ?m)) (desc n)))
58   (printout t "Inicial ->" ?s ": " (calc-coste ?s ?m) crlf)
59 )

```

Esta regla solo se ejecutará una vez al principio, se encarga de crear la lista que representa la matriz de distancias para las n ciudades, los hechos que controlarán la generación de todos los intercambios y la solución inicial, que será recorrer las ciudades en orden.

La siguiente regla será la que haga la exploración en cada iteración (es un decir) de todos los intercambios posibles.

```

60 (defrule HC-paso1
61   (pos ?i)
62   (pos ?j&:(> ?j ?i))
63   (solucion (camino $?s) (coste ?c) (desc n))
64   (mat-dist (dist $?m))
65   =>
66   (bind ?vi (nth$ ?i ?s))
67   (bind ?vj (nth$ ?j ?s))
68   (bind ?sm (delete$ ?s ?i ?i))
69   (bind ?sm (insert$ ?sm ?i ?vj))
70   (bind ?sm (delete$ ?sm ?j ?j))
71   (bind ?sm (insert$ ?sm ?j ?vi))
72   (bind ?nc (calc-coste ?sm ?m))
73   (assert (solucion (camino ?sm) (coste ?nc) (desc s)))
74 )

```

Esta regla simplemente se instancia con todos los intercambios posibles y crea una solución descendiente de la actual. Esto nos llenará la memoria de hechos de este tipo para todas las veces que se pueda instanciar. No hace falta que nos preocupemos de si la solución es mejor que la actual ya que utilizaremos otra regla para seleccionar la que tenga el mejor coste de todas las generadas. Esto lo haremos con la siguiente regla:

```

75 (defrule HC-paso2
76   (declare (salience -10))
77   (solucion (camino $?s) (coste ?c) (desc s))
78   (forall
79     (solucion (coste ?oc) (desc s))
80     (test (<= ?c ?oc)))
81   ?solact <- (solucion (coste ?cs&:(< ?c ?cs)) (desc n))
82   =>

```

```

83  (modify ?solact (camino ?s) (coste ?c))
84  (printout t "->" ?s ": " ?c crlf)
85  )

```

Simplemente decimos que queremos un solución descendiente que tenga el mínimo coste y su coste sea mejor que el de la solución actual. Esta regla queremos que se ejecute una vez hemos ejecutado todas las instanciaciones posibles de la regla anterior, por eso le ponemos menos prioridad.

Sabremos que hemos acabado la búsqueda cuando ninguna de las reglas anteriores se pueda ejecutar. Podemos incluir una regla que nos imprima la solución cuando eso suceda:

```

86  (defrule HC-es-fin
87    (declare (salience -20))
88    (solucion (camino $?s) (coste ?c) (desc n))
89    =>
90    (printout t "Final ->" ?s ": " ?c crlf)
91  )

```

Al darle prioridad inferior sabremos que se ejecutará al final de todo.

En este caso, hemos puesto las prioridades de esta manera ya que queremos que la exploración se comporte como el Hill Climbing original, pero podríamos jugar con ellas o con la estrategia de resolución de conflictos del motor de inferencia para reproducir otros comportamientos. Si por ejemplo diéramos a las dos reglas la misma prioridad, cada vez que se ejecutara la regla del segundo paso se actualizaría la solución actual, cambiando el punto desde el que se sigue explorando. Si usamos una estrategia de resolución de conflictos en profundidad tendríamos un hill climbing que se movería en cada paso al primer movimiento que mejora la solución actual, en lugar de escoger el mejor de todos que es lo que nos dan las prioridades puestas.

Limpiando un poco la memoria

Las reglas tal como están hacen exactamente lo que queremos, pero podemos darnos cuenta de un defecto. Vamos llenando la memoria de trabajo con todas las soluciones intermedias que vamos calculando y evidentemente llenar la memoria de trabajo incrementa el trabajo del motor de inferencia.



De hecho la unificación y la detección de las reglas a aplicar se hace de manera bastante eficiente en los motores de razonamiento hacia adelante (algoritmo de RETE), pero para obtener esta eficiencia se deben construir estructuras que permiten actualizar las unificaciones y eso incrementa el gasto en memoria.

Un primer paso sencillo es no generar aquellas soluciones que sean peores que la actual, podemos modificar la regla **HC-paso1** simplemente cambiando el assert final por:

```

(if (< ?nc ?c)
  then (assert (solucion (camino ?sm) (coste ?nc) (desc s))))

```

Otra estrategia para no llenar la memoria de trabajo es limpiar todos los sucesores una vez hemos obtenido la mejor solución de entre ellos. Para hacer esto deberemos pensar en nuestro algoritmo en dos pasos diferenciados, las reglas que buscan y las reglas que hacen la limpieza de la memoria

de trabajo. Para ello utilizaremos dos hechos de control (`buscar`) y (`limpiar`) que utilizaremos para activar las reglas que hacen cada parte. Cuando acaben unas pasarán el control a las otras simplemente añadiendo y quitando los correspondientes hechos de control.

De esta manera, la regla que inicializa la búsqueda tendrá que añadir el hecho de control que activa las reglas de búsqueda:

```

1 (defrule init
2   (num-ciudades ?x)
3   =>
4   (bind ?m (create$))
5   (loop-for-count (?i 1 (/ (* ?x (- ?x 1)) 2))
6     do
7       (bind ?m (insert$ ?m ?i (+ (mod (random) 50) 1)))
8     )
9   (assert (mat-dist (dist ?m)))
10  (loop-for-count (?i 1 ?x)
11    do
12      (assert (pos ?i))
13    )
14  (bind ?s (create$))
15  (loop-for-count (?i 1 ?x)
16    do
17      (bind ?s (insert$ ?s ?i ?i))
18    )
19  (assert (solucion (camino ?s) (coste (calc-coste ?s ?m)) (desc n)))
20  (assert (busca))
21  (printout t "Inicial ->" ?s ": " (calc-coste ?s ?m) crlf)
22 )

```

Tendremos las tres reglas que realizan la búsqueda que estarán controladas por el hecho `busca`:

```

23 (defrule HC-paso1
24   (busca)
25   (pos ?i)
26   (pos ?j&:(> ?j ?i))
27   (solucion (camino $s) (coste ?c) (desc n))
28   (mat-dist (dist $m))
29   =>
30   (bind ?vi (nth$ ?i ?s))
31   (bind ?vj (nth$ ?j ?s))
32   (bind ?sm (delete$ ?s ?i ?i))
33   (bind ?sm (insert$ ?sm ?i ?vj))
34   (bind ?sm (delete$ ?sm ?j ?j))
35   (bind ?sm (insert$ ?sm ?j ?vi))
36   (bind ?nc (calc-coste ?sm ?m))
37   (if (< ?nc ?c)
38     then (assert (solucion (camino ?sm) (coste ?nc) (desc s))))
39 )
40
41 (defrule HC-paso2

```

```

42 (declare (salience -10))
43 ?h <-(busca)
44 ?solact <- (solucion (desc n))
45 (solucion (camino $?s) (coste ?c) (desc s))
46 (forall
47   (solucion (coste ?oc) (desc s))
48   (test (<= ?c ?oc)))
49 =>
50 (modify ?solact (camino ?s) (coste ?c))
51 (printout t "->" ?s ": " ?c crlf)
52 (assert (limpia))
53 (retract ?h)
54 )
55
56 (defrule HC-es-fin
57   (declare (salience -20))
58   (busca)
59   (solucion (camino $?s) (coste ?c) (desc n))
60   =>
61   (printout t "Final ->" ?s ": " ?c crlf)
62 )

```

Hay que observar que solo la regla **HC-paso2** es la que tiene que pasar el control a la parte que limpia la memoria de trabajo. También hay que observar que la condición de esta regla ya no exige que el coste del mejor descendiente sea también mejor que la solución actual.

De hecho la razón real por la que está esa condición en el primer conjunto de reglas es un poco mas complicada de lo que aparentemente pueda parecer. Si no exigimos esa condición, la regla puede entrar en un bucle infinito debido a su funcionamiento. Ésta modifica el hecho que corresponde a la solución actual con la mejor solución (lo lógico), eso hace que la regla pueda volver a activarse. Como en este conjunto de reglas vamos a eliminar a continuación las soluciones descendientes ya no hay peligro de bucle infinito.

Para la limpieza de los hechos intermedios podemos usar el siguiente conjunto de reglas:

```

63 (defrule HC-clean
64   (limpia)
65   ?h <- (solucion (desc s))
66   =>
67   (retract ?h)
68 )
69
70 defrule HC-reinit
71   ?h <- (limpia)
72   (not (solucion (desc s)))
73   =>
74   (retract ?h)
75   (assert (busca))
76 )

```

La primera simplemente se va instanciando con los hechos y eliminándolos. Cuando ya no quedan hechos la siguiente regla pasa el control a las reglas que hacen la búsqueda.

Este ejemplo ilustra como se pueden dividir las diferentes partes de un algoritmo descrito mediante reglas e ir pasando el control de unas partes a otras. En el caso de que lo que haga cada parte sea relativamente sencillo este mecanismo es bastante práctico.



Si nos enfrentamos ante algo mas complejo, lo mejor es utilizar el mecanismo de módulos de CLIPS. Por un lado será mas eficiente y el control será mas sencillo. Por otro, no nos hará falta eliminar de la memoria de trabajo los hechos que se generen que sean privados del módulo, ya que son borrados automáticamente una vez un módulo acaba su ejecución.

Ahora con objetos

Simplemente para mostrar como se puede utilizar el sistema de objetos de CLIPS vamos a reescribir el problema transformando los hechos definidos como `deftemplates` a objetos. La ventaja que tienen los objetos es que tenemos primitivas que permiten consultar los objetos de la base de hechos al estilo de los patrones que usamos en las reglas, lo cual nos facilitará la limpieza de los hechos temporales. Además, se puede estructurar mejor la información y hacer las reglas un poco mas claras dejando el trabajo a los objetos.



Este es un buen momento para repasar como se definen los objetos en CLIPS, los mecanismos para tratarlos dentro de las reglas y la definición e invocación de métodos.

Empezaremos definiendo las clases que representarán la información del problema:

```

1 (defclass mat-dist
2   (is-a USER)
3   (role concrete)
4   (pattern-match reactive)
5   (slot nciu)
6   (multislot dist)
7 )
8
9 (defclass solucion
10  (is-a USER)
11  (role concrete)
12  (pattern-match reactive)
13  (multislot camino)
14  (slot coste (type INTEGER))
15  (slot desc)
16 )

```

No hay mayor diferencia respecto a como lo hemos definido en las versiones anteriores. En este caso hemos incluido el numero de ciudades dentro de la matriz de distancias.

Para tratar con las diferentes operaciones que realizamos las definiremos ahora como `message-handlers` ahorrándonos pasar algunos parámetros y encapsulando la modificación de la solución.

```

17 (defmessage-handler mat-dist calc-dist (?i ?j)
18   (if (< ?i ?j)
19     then (bind ?pm ?i) (bind ?pn ?j)

```

```

20   else (bind ?pm ?j) (bind ?pn ?i)
21   )
22   (bind ?off 0)
23   (loop-for-count (?i 1 (- ?pm 1))
24   do
25     (bind ?off (+ ?off (- ?self:nciu ?i)))
26   )
27   (nth$ (+ ?off (- ?pn ?pm)) ?self:dist)
28 )
29
30 (defmessage-handler solucion calc-coste (?m)
31   (bind ?self:coste 0)
32   (loop-for-count (?i 1 (- (length$ ?self:camino) 1))
33   do
34     (bind ?self:coste
35       (+ ?self:coste (send ?m calc-dist
36                         (nth$ ?i ?self:camino)
37                         (nth$ (+ ?i 1) ?self:camino))))
38   )
39   (bind ?self:coste
40     (+ ?self:coste (send ?m calc-dist
41                         (nth$ 1 ?self:camino)
42                         (nth$ (length$ ?self:camino) ?self:camino))))
43   )
44
45 (defmessage-handler solucion imprime-sol ()
46   (printout t ?self:camino ": " ?self:coste crlf )
47   )
48
49 (defmessage-handler solucion intercambia (?i ?j ?m)
50   (bind ?vi (nth$ ?i ?self:camino))
51   (bind ?vj (nth$ ?j ?self:camino))
52   (bind ?sm (delete$ ?self:camino ?i ?i))
53   (bind ?sm (insert$ ?sm ?i ?vj))
54   (bind ?sm (delete$ ?sm ?j ?j))
55   (bind ?sm (insert$ ?sm ?j ?vi))
56   (bind ?self:camino ?sm)
57   (send ?self calc-coste ?m)
58 )

```

El primer `message-handler` retorna la distancia entre dos ciudades, el segundo calcula el coste de un camino, el tercero imprime la información de una solución y el cuarto hace el intercambio entre dos ciudades y actualiza el coste del camino.



El lenguaje orientado a objetos de CLIPS no solo se puede utilizar para definir los conceptos del dominio del programa. Utilizar técnicas de programación orientada a objetos puede ayudar a desarrollar programas en CLIPS permitiendo el encapsulamiento de datos. Pensad también que al ser CLIPS un lenguaje débilmente tipado se pueden sobrecargar los métodos y aplicar técnicas de genericidad. El mecanismo de herencia puede ayudar además para simplificar la programación de los métodos.

Ahora la regla que inicializa la búsqueda será algo diferente dado el cambio de representación:

```

59 (defrule init
60   (num-ciudades ?x)
61 =>
62   (bind ?m (create$))
63   (loop-for-count (?i 1 (/ (* ?x (- ?x 1)) 2))
64     do
65       (bind ?m (insert$ ?m ?i (+ (mod (random) 50) 1)))
66     )
67   (bind ?md
68     (make-instance (gensym) of mat-dist (dist ?m) (nciu ?x)))
69   (loop-for-count (?i 1 ?x)
70     do
71       (assert (pos ?i))
72     )
73   (bind ?s (create$))
74   (loop-for-count (?i 1 ?x)
75     do
76       (bind ?s (insert$ ?s ?i ?i))
77     )
78   (bind ?sol
79     (make-instance (gensym) of solucion
80                     (camino ?s) (desc n)))
81   (send ?sol calc-coste ?md)
82   (printout t "Inicial -> ")
83   (send ?sol imprime-sol)
84 )

```

Ahora las instancias se crean mediante `make-instance`, la función `(gensym)` genera un símbolo nuevo cada vez que se la llama y la usamos para generar los nombres de las instancias.

Las reglas que hacen la búsqueda también cambian ligeramente para adaptarnos a la representación con objetos. Cambiamos las condiciones utilizando la construcción `object` y modificamos y creamos las soluciones usando las funciones que dan los objetos:

```

85 (defrule HC-paso1
86   (pos ?i)
87   (pos ?j&:(> ?j ?i))
88   ?s <- (object (is-a solucion) (desc n))
89   ?m <- (object (is-a mat-dist))
90   =>
91   (bind ?ns (duplicate-instance ?s to (gensym)))
92   (send ?ns put-desc s)
93   (send ?ns intercambia ?i ?j ?m)
94   (if (< (send ?s get-coste) (send ?ns get-coste))
95     then (send ?ns delete))
96 )

```

Las soluciones descendientes las creamos duplicando el objeto que tiene la solución actual y modificando el camino. En este caso cada vez que se crea un objeto, este se almacena en la memoria de trabajo, por lo que debemos borrarlo si tiene un coste mayor. De esta manera obtenemos el mismo efecto que en la solución anterior.

La regla que se queda con la mejor solución tampoco es muy diferente:

```

97 (defrule HC-paso2
98   (declare (salience -10))
99   ?solmejor <- (object (is-a solucion) (coste ?c) (desc s))
100   (forall
101     (object (is-a solucion) (coste ?oc) (desc s))
102     (test (<= ?c ?oc)))
103   ?solact <- (object (is-a solucion) (coste ?cs&:(< ?c ?cs)) (desc n))
104   =>
105   (send ?solmejor put-desc n)
106   (send ?solact delete)
107   (do-for-all-instances ((?sol solucion)) (eq ?sol:desc s)
108     (send ?sol delete))
109   (send ?solmejor imprime-sol)
110 )

```

En este caso cambiamos el valor del slot `desc` a `n` y borramos la solución actual. También aprovechamos una de las funciones que permiten hacer consultas a la base de objetos para eliminar todas las soluciones descendientes. Podríamos haber adoptado sin ningún problema el mismo método que en la solución anterior, pero esta opción es mas eficiente.

Finalmente tenemos la regla que imprime la solución cuando acaba la búsqueda:

```

111 (defrule HC-es-fin
112   (declare (salience -20))
113   ?sol <- (object (is-a solucion) (desc n))
114   =>
115   (printout t "Final -> ")
116   (send ?sol imprime-sol)
117 )

```



Todavía se pueden hacer mas modificaciones a la solución. Por ejemplo, se podrían hacer dos especializaciones de la clase solución, una para la solución actual y otra para las descendientes, eliminando así la necesidad de tener el slot `desc`. También se podría incluir la matriz de distancias en el objeto solución, evitando así que tener que añadirla a las condiciones de las reglas.

2.3 El sistema experto en reparación de automóviles

Uno de los ejemplos que incluye la distribución de CLIPS es un sistema experto para el diagnóstico de problemas en un automóvil. El sistema es muy sencillo y solo permite dar un número limitado de respuestas, pero es ilustrativo de como se puede controlar el flujo de preguntas y la generación de hipótesis en un programa basado en reglas. Podeis acceder al código del programa en <http://www.lsi.upc.edu/~bejar/ia/material/laboratorio/clips/auto.clp>.

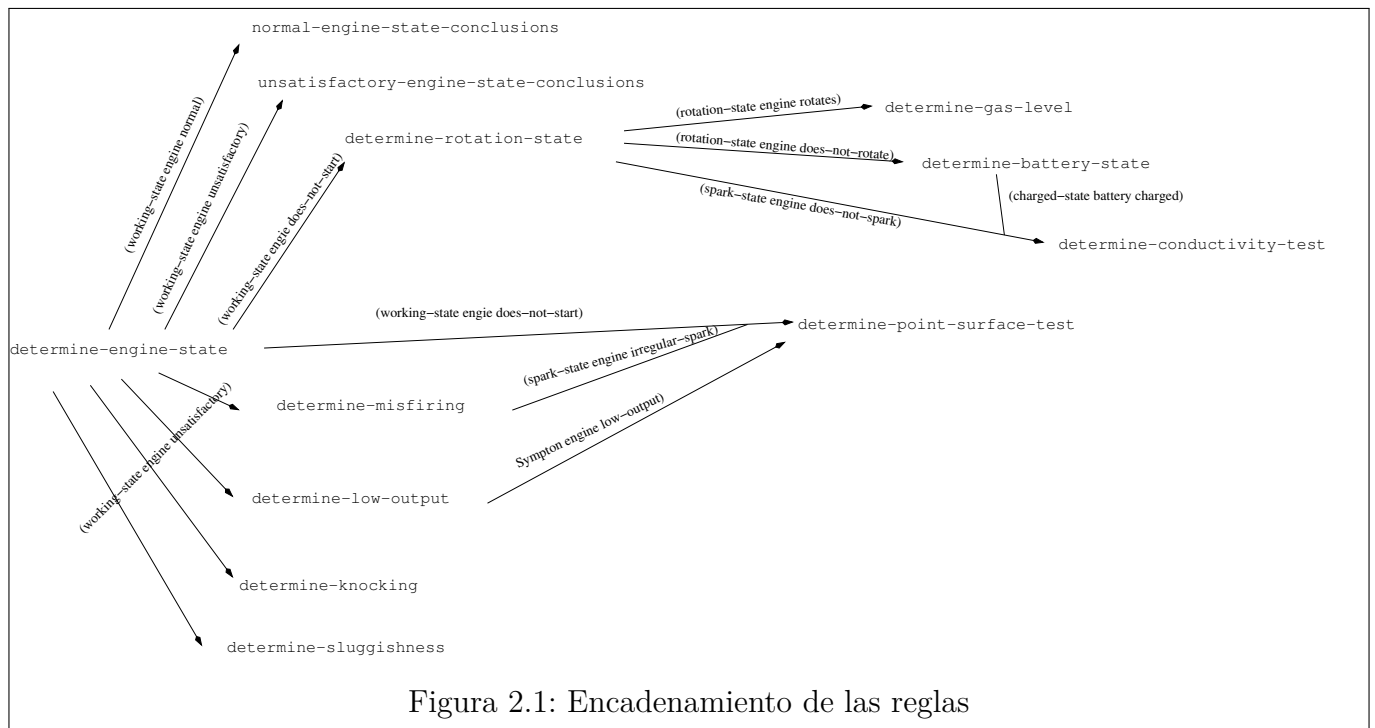


Figura 2.1: Encadenamiento de las reglas

El control está basado en un conjunto de hechos que se van añadiendo en la memoria de trabajo y que permiten que las reglas se puedan ir ejecutando en un orden establecido para llegar a los diferentes diagnósticos que se pueden dar. Como el problema es bastante sencillo se utilizan *unordered facts*, para un problema más complejo haría falta estructurar más la información.

Los hechos que se utilizan son los siguientes:

- (working-state engine normal|unsatisfactory|does-not-start)
- (rotation-state engine rotates|does-not-rotate)
- (spark-state engine irregular-spark|does-not-spark)
- (symptom engine low-output|not-low-output)
- (charge-state battery charged)

Siguiendo la estructura que se ha determinado para el diagnóstico se van haciendo preguntas y se van añadiendo hechos a partir de las respuestas que hacen que la secuencia de ejecución de las reglas lleven a un diagnóstico específico. En la figura 2.1 se puede ver los diferentes encadenamientos entre las reglas del programa.

Antes de entrar en detalle con las reglas cabe comentar las funciones que se utilizan para hacer las preguntas al usuario en las reglas.

```

1 (defun ask-question (?question $?allowed-values)
2   (printout t ?question)
3   (bind ?answer (read))
4   (if (lexemep ?answer)
5       then (bind ?answer (lowercase ?answer)))
6   (while (not (member ?answer ?allowed-values)) do
7     (printout t ?question)
8     (bind ?answer (read))

```

```

9      (if (lexemep ?answer)
10         then (bind ?answer (lowercase ?answer))))
11      ?answer)
12
13 (deffunction yes-or-no-p (?question)
14   (bind ?response (ask-question ?question yes no y n))
15   (if (or (eq ?response yes) (eq ?response y))
16       then TRUE
17       else FALSE))

```

La primera función (**ask-question**) recibe el texto de una pregunta como primer parámetro y un número no acotado de parámetros que corresponderán a las respuestas aceptables. El segundo parámetro es una variable que acepta múltiples valores y es donde se guardarán como una lista los parámetros extra que recibamos.

Para imprimir la pregunta se utiliza el operador **printout** que recibe un canal (**t** representa la salida estándar) y una cadena. Para leer la entrada se utiliza el operador **read** que lee de teclado. En CLIPS al leer se interpreta la entrada para asignarla al tipo correspondiente, de ahí que se comprueba si la entrada es una cadena o un símbolo (**lexemp**). La función leerá hasta que reciba una respuesta que este en la lista que se ha pasado como parámetro y retornará el valor leído.

La función **yes-or-no-p** solo admite como respuestas (**yes no y n**) y retorna cierto o falso dependiendo de si la respuesta es afirmativa o negativa.

En el programa se distinguen tres grupos de reglas. Las que solamente deducen hechos y no preguntan al usuario, las que preguntan al usuario y las que presentan el sistema y escriben el resultado.

Estas últimas reglas se controlan mediante la **salience** y la aparición en la memoria de trabajo del hecho que indica que se ha obtenido una respuesta (**repair ?**).

```

1  ;;*****
2  ;;* STARTUP AND REPAIR RULES *
3  ;;*****
4
5  (defrule system-banner ""
6    (declare (salience 10))
7    =>
8    (printout t crlf crlf)
9    (printout t "The Engine Diagnosis Expert System")
10   (printout t crlf crlf))
11
12 (defrule print-repair ""
13   (declare (salience 10))
14   (repair ?item)
15   =>
16   (printout t crlf crlf)
17   (printout t "Suggested Repair:")
18   (printout t crlf crlf)
19   (format t " %s%n%n%n" ?item))

```

Para el resto de reglas se utilizan la presencia o la ausencia de hechos en la memoria de trabajo para dispararlas. Por ejemplo, todas las reglas que hacen el diagnóstico tienen la condición (**not**

(repair ?)) para que se dejen de ejecutar en el momento de que una de las reglas que llega a la conclusión final deduce la solución.

Será el comportamiento del algoritmo que realiza el diagnóstico el que nos permita saber cuando activar las reglas y por lo tanto que condiciones hemos de poner en cada una de ellas. Por ejemplo, la regla inicial del proceso de diagnóstico es:

```

1  (defrule determine-engine-state ""
2    (not (working-state engine ?))
3    (not (repair ?))
4    =>
5    (if (yes-or-no-p "Does the engine start (yes/no)? ")
6        then
7        (if (yes-or-no-p "Does the engine run normally (yes/no)? ")
8            then (assert (working-state engine normal))
9            else (assert (working-state engine unsatisfactory)))
10       else
11       (assert (working-state engine does-not-start))))

```

Todas las demás reglas dependen directa o indirectamente del hecho (working-state engine ?), por lo que será la primera en ejecutarse y preguntar por el estado del motor.

También se utiliza como mecanismo de control la salience, de manera que nos aseguremos de que ciertas reglas se ejecuten primero, como por ejemplo:

```

1  (defrule unsatisfactory-engine-state-conclusions ""
2    (declare (salience 10))
3    (working-state engine unsatisfactory)
4    =>
5    (assert (charge-state battery charged))
6    (assert (rotation-state engine rotates)))

```

que se ejecutarán por ejemplo antes que:

```

1  (defrule determine-sluggishness ""
2    (working-state engine unsatisfactory)
3    (not (repair ?))
4    =>
5    (if (yes-or-no-p "Is the engine sluggish (yes/no)? ")
6        then (assert (repair "Clean the fuel line.")))

```

o que ciertas reglas se ejecuten cuando ninguna mas se pueda ejecutar:

```

1  (defrule no-repairs ""
2    (declare (salience -10))
3    (not (repair ?))
4    =>
5    (assert (repair "Take your car to a mechanic.")))

```

En los mecanismos de control se puede jugar con la presencia o la ausencia de hechos, haciendo que la ejecución complete la información que tenemos del problema o siga un camino que permita deducir hechos derivados de los que ya tenemos, por ejemplo:

```

1  (defrule determine-battery-state ""
2    (rotation-state engine does-not-rotate)
3    (not (charge-state battery ?))
4    (not (repair ?))
5    =>
6    (if (yes-or-no-p "Is the battery charged (yes/no)? ")
7        then
8        (assert (charge-state battery charged))
9        else
10       (assert (repair "Charge the battery. "))
11       (assert (charge-state battery dead))))
12
13 (defrule determine-conductivity-test ""
14   (working-state engine does-not-start)
15   (spark-state engine does-not-spark)
16   (charge-state battery charged)
17   (not (repair ?))
18   =>
19   (if (yes-or-no-p "Is the conductivity test for the ignition coil
20                                   positive (yes/no)? ")
21       then
22       (assert (repair "Repair the distributor lead wire. "))
23       else
24       (assert (repair "Replace the ignition coil. "))))

```

Podeis jugar con el programa viendo que conjuntos de respuestas llevan a cada uno de los diagnósticos. Pensad que con las reglas estamos estableciendo el grafo que conecta los diferentes hechos del problema con la solución. Este grafo puede ser muy sencillo o muy complejo, solo hemos de determinar qué caminos son los que queremos que las reglas nos representen.

Por ejemplo, supongamos que hacemos la siguiente interacción con el programa:

The Engine Diagnosis Expert System

```

Does the engine start (yes/no)? no
Does the engine rotate (yes/no)? yes
What is the surface state of the points (normal/burned/contaminated)? normal
Does the tank have any gas in it (yes/no)? no

```

Suggested Repair:

```
Add gas.
```

Si seguimos la ejecución del programa, la primera regla que se ejecutará será **system-banner** ya que tiene la mayor salience y no tiene condiciones de ejecución. Después de esta se ejecutará la regla

`determine-engine-state` que es la que inicia las preguntas para llegar al diagnóstico. Si nos fijamos en la agenda veremos que tendremos siempre la regla `no-repairs` que está esperando a que ninguna otra regla de más prioridad se pueda ejecutar.

Al responder `no` a la pregunta se añadirá el hecho (`working-state engine does-not-start`). Este hecho hace que la regla `determine-rotation-state` se pueda ejecutar, realizando la pregunta sobre el estado de la rotación del motor. Esta regla añade dos hechos al responderla afirmativamente, (`rotation-state engine rotate`) y (`spark-state engine irregular-spark`). Esto hace que haya dos reglas con la misma prioridad que se puedan ejecutar en este momento, `determine-point-surface-state` y `determine-gas-level`.

En este caso el orden de ejecución dependerá de la estrategia de resolución de conflictos que tenga activada el motor de inferencia. Si observáis la agenda durante la ejecución del programa y cambiáis la estrategia vereis que el orden de las reglas en esta cambia.

Suponiendo que la estrategia de resolución de conflictos escoge la regla `determine-point-surface-state` y respondemos que el estado de la superficie de las bujía es normal no tendremos nuevos hechos en la base de hechos, por lo que se ejecutará la regla `determine-gas-level`. Al responder negativamente a la pregunta de esta regla se añade a la base de hechos el hecho (`repair "Add gas."`) que es el que hace que se active la regla que escribe la solución, finalizándose así la ejecución del programa.

Obviamente, con un conjunto diferente de respuestas seguiríamos otro encadenamiento de reglas y obtendríamos otro resultado.