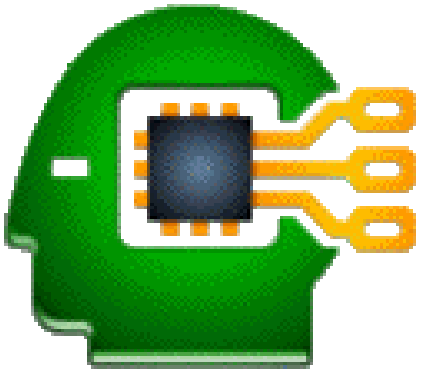


Inteligencia Artificial

Representación del conocimiento y razonamiento

ROBOTIKA & SISTEMA AUTONOMOEN IKERKETA TALDEA

eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea

Hechos

- Un **hecho** es una forma básica de **representar la información**. Puede tener un campo o varios, de tipo **numérico**, **simbólico** o **String**.

- **CLIPS diferencia entre mayúsculas y minúsculas**.

- Existen dos tipos de hechos; **ordenados** y **no ordenados (en la parte 2)**:
  - En un hecho, al principio se pone el ***nombre_del_hecho:*** *color, padre_de …*
  - En los **hechos ordenados**, **es importante el orden de los elementos o símbolos.**

(padre_de Juan Pepe) **(pacman 3 8 2 3 30)**

Hechos no ordenados

- En los hechos no ordenados **cada campo tiene su nombre y valor**.

(padre_de (hijo Juan) (padre Pepe))

(pacman (mundo “ ### # ##”)(posicionP 3 8)(fantasmaP 2 3)(comida 30))

- Los hechos no ordenados **proporcionan al usuario la habilidad de abstraerse de la estructura** del hecho, asignando un nombre a cada campo del mismo.

- Para los **hechos no ordenados** CLIPS **requiere la definición** previa de su **plantilla**, por medio del constructor ***deftemplate***, para especificar el modelo del hecho.

- El constructor ***deftemplate*** crea una plantilla o patrón que se usa para acceder por su nombre a los campos (***slots/multislots***) de un hecho no ordenado. **Este constructor es parecido a la definición de un registro en lenguajes como Ada o las estructuras de C**.

Hechos no ordenados: plantillas

- Para definir una plantilla con el constructor ***deftemplate*** se especifica el nombre de cada campo

```
(deftemplate datos-persona ; nombre de la plantilla
    (multislot nombre) ; este campo puede tener varios valores
    (multislot apellidos)
    (slot edad) ; este campo solo tendrá un valor
    (slot peso)
    (slot altura)
    (multislot presion-arterial)
)
```

- Comandos:
  - ***(list-deftemplates)*** listamos los **nombres de las plantillas** definidas
  - ***(ppdeftemplate <nombre>)*** muestra **el contenido** de la plantilla
  - ***(undeftemplate <nombre>)*** permite **eliminar la definición**, siempre que no existan hechos en la base de hechos que sigan esa plantilla

Hechos no ordenados: plantillas

- **Los hechos ordenados permiten juntar trozos de información**. Un ejemplo podría ser el estado de forma de una persona:

Hechos ordenados

```
(edad Samuel 20)
(peso Samuel 80)
(altura Samuel 188)
(presion-arterial Samuel 80 130)

(edad Eva 23)
(peso Eva 50)
(altura Eva 155)
(presion-arterial Eva 60 120)
```

Plantilla para hecho desordenado

```
(deftemplate datos-persona
    (multislot nombre)
    (slot edad)
    (slot peso)
    (slot altura)
    (multislot presion-arterial)
)
```

- Cuando hay muchos datos relacionados, es más cómodo utilizar la plantilla:

```
(assert (datos-persona (nombre Samuel)(edad 20)(peso 80)(altura 188)(presion-arterial 80 130)))
(assert (datos-persona (peso 50)(nombre Eva)(edad 23)(altura 155)(presion-arterial 60 120)))
```

- **No hace falta que los campos estén ordenados!**.

Hechos no ordenados: atributos de las plantillas

- Una plantilla **permite definir** además del nombre, el **tipo**, los **valores por defecto** y el **rango** de sus *slots* o *multislots*.
- Los tipos posibles son: **SYMBOL**, **STRING**, **LEXEME**, **NUMBER**, **INTEGER** y **FLOAT**.
- Para definir el tipo de dato **que admite** un slot/multislot, se usa ***type***.
  - **LEXEME** → **equivale** a especificar **SYMBOL** y **STRING** conjuntamente
  - **NUMBER** → **equivale** a especificar **INTEGER** y **FLOAT**
  - **?VARIABLE** → puede tomar **cualquier tipo de dato** **(por defecto)**
- ***allowed*:** Especifica los valores concretos permitidos; ***allowed-symbols, allowed-strings, allowed-lexemes, allowed-integers, allowed-floats, allowed-numbers*** y ***allowed-values***

```
(deftemplate persona
      (multislot nombre (type SYMBOL))
      (slot edad (type INTEGER))
      (slot altura (type SYMBOL)(allowed-symbols alto bajo)))
```

- **Por defecto: *(allowed-values ?VARIABLE)* → indica que cualquier valor es legal**

Hechos no ordenados: atributos de las plantillas

- ***range:*** **permite restringir los valores de un tipo numérico a un rango determinado**. Los límites pueden ser un valor numérico o ?VARIABLE (infinito).

(range ?VARIABLE 3) → representa el rango -∞ .. 3
(range 14 ?VARIABLE) → representa el rango 14 .. +∞
(range ?VARIABLE ?VARIABLE) → representa el rango -∞ .. +∞ **(por defecto)**

```
(deftemplate persona
    (multislot nombre (type SYMBOL))
    (slot edad (type INTEGER)(range 0 125))
```

- ***cardinality:*** **Permite especificar el número mínimo y máximo de valores que un *multislot* puede contener**. Ambos límites pueden ser un entero positivo ó ?VARIABLE. **Por defecto** se supone **?VARIABLE** para ambos límites.

```
(deftemplate equipo-voleibol
   (slot nombre-equipo (type STRING))
   (multislot jugadores (type STRING) (cardinality 6 6)) → debe tener 6 jugador@s sí o sí
   (multislot sustitutos (type STRING) (cardinality 0 2))) → puede haber 0 o 2 sustitutos
```

Hechos no ordenados: atributos de las plantillas

- ***default:*** **Permite especificar un valor por defecto (default <especificación>)**
- La <***especificación***> puede ser, ***?DERIVE, ?NONE*** o una **expresión (valor):**
  - **?DERIVE: se deriva un valor para el *slot* que satisfaga todos los atributos del *slot*.** Si en un *slot* no se especifica nada acerca de ***default***, se supondrá (**default ?DERIVE**).
    - Para **String → " ",** para **Integer → 0,** para **Symbol → nil,** para **Float → 0.0**
  - **?NONE: hay que darle un valor obligatoriamente**

```
(deftemplate datos-persona
    (slot nombre (type STRING) (default ?DERIVE))
    (slot apellido (type STRING) (default ?NONE))
    (slot edad (type FLOAT) (default (* 2.0 3.5)) (range 0.0 120.0)) )
```

```
CLIPS> (assert (datos-persona (apellido "Jauregi")))
<Fact-1>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (datos-persona (nombre "") (apellido "Jauregi") (edad 7.0))
For a total of 2 facts.
```

Hechos: atributos de las plantillas

- **Ejemplos**

```
(deftemplate ejemplo
    (slot a) → por defecto SYMBOL, y el valor será nil
    (slot b (type INTEGER)) → el valor por defecto para INTEGER es 0
    (slot c (allowed-values rojo verde azul)) → el valor por defecto será el primero
    (multislot d) → por defecto contendrá el valor ()
    (multislot e (cardinality 2 2)(type FLOAT)(range 3.5 10.0))) → el valor por defecto será el min
```

- Al introducir un hecho en la MT sin dar ningún valor a ninguno de sus *slots*, el hecho realmente contendría lo siguiente: **(assert (ejemplo))**

(ejemplo (a nil) (b 0) (c rojo) (d) (e 3.5 3.5))

```
(deftemplate ejemplo
    (slot a (default 3))
    (slot b (default (+ 3 4)))
    (multislot c (default a b e))
    (multislot d (default (+ 1 2) (+ 3 4))))
```

- Al introducir un hecho en la MT sin dar ningún valor a ninguno de sus *slots*, el hecho realmente contendría lo siguiente: **(assert (ejemplo))**

(ejemplo (a 3) (b 7) (c a b e) (d 3 7))

Hechos: definición de hechos iniciales (deffacts)

- Template para ***estudiante:***

```
(deftemplate estudiante
    (multislot nombre)
    (slot fuma (allowed-values si no))
    (slot alojado (allowed-values si no)(default no))
)
```

```
(deffacts estudiantes "Todos los estudiantes iniciales"
    (estudiante (nombre Juan)(fuma no)(alojado no))
    (estudiante (nombre Pepe)(fuma si) (alojado no))
    (estudiante (nombre Luisa)(fuma no) (alojado no))
    (estudiante (nombre Pedro)(fuma no) (alojado no))
)
```

Hechos: operaciones

- Las acciones que se pueden realizar sobre los hechos:
  - Acciones que modifican la MT: lista de hechos
    - para **insertar** hechos: ***(assert <hecho>+)***
    - ***(duplicate <especificador-hecho> <slot>*)*** → **Nota2**
    - Para **borrar** hechos: ***(retract <especificador-hecho>)*** → **Nota1**
    - Para **modificar** elementos: ***(modify <especificador-hecho> <slot>*)*** → **Nota2**
  - donde <especificador-hecho> puede ser:
    - **una variable previamente ligada a la dirección del hecho** a duplicar, borrar o modificar. (**esto se analizará con las reglas**)
    - **un índice de hecho** (aunque no se conoce durante la ejecución de un programa).

Nota1: Puede utilizarse el símbolo '*****' con el comando ***retract*** para eliminar todos los hechos

Nota2: **Solo con hechos NO ORDENADOS**

Reglas

- Las reglas **permiten operar con los hechos.**
- Una regla consta de un **antecedente** -también denominado parte “si” o parte izquierda de la regla (LHS) - y de un **consecuente** -también denominado parte “entonces” o parte derecha de la regla (RHS).
- El antecedente está formado por un conjunto de condiciones -también denominadas **elementos condicionales** (EC) que deben satisfacerse para que la regla sea aplicable
- El consecuente de una regla es un conjunto de acciones a ser ejecutadas cuando la regla es aplicable.

```
(defrule regla
   (tiempo nublado)          } parte LHS o antecedente
   (tiempo no viento)        }
=>
   (assert (coger paraguas)) } parte RHS o consecuente
)
```

Tipos de elementos condicionales (LHS)

- ***pattern:*** Colección de restricciones de campos, comodines, y variables que se usan para restringir el conjunto de hechos o instancias que satisfacen el ***pattern***.

  - **Reglas: restricciones literales**:

```
(altitud es 1000 metros)
(habitacion (numero 23)(plazas-libres 3)(ocupantes eka1 eka2))
```

BC

```
(defrule monte
   (altitud es 1000 metros)
=>
   (assert (monte alto)))
```

  - **Reglas: con variables simples y multicampo**:

```
(altitud es ?x metros)
(habitacion (numero ?y)(plazas-libres 3))
(habitacion (numero ?y)(plazas-libres ?y))
(habitacion (numero ?y)(plazas-libres 3)(ocupantes $?z))
```

MT

```
(initial-facts)
(altitud es 1000 metros)
```

  - **Reglas: con comodines simples y multicampo**:

```
(altitud es ? metros)
(habitacion (numero ?)(plazas-libres 3))
(habitacion (numero ?)(plazas-libres 3)(ocupantes $?))
```

Los ***comodines*** permiten que encaje cualquier valor **sin atraparlo**.

- ***?*** para un solo campo
- ***$?*** para un multicampo

Tipos de elementos condicionales (LHS)

- Se pueden especificar **restricciones al comparar un patrón** (operadores lógicos). El orden de prioridad es el presentado a continuación:

```
Negación: ~
Disyunción: |
Conjunción: &
```

```
(habitación (plazas-libres ~0))
(habitación (plazas-libres 1|2|3))
(habitación (plazas-libres ~0 & ~4))
(habitación (plazas-libres ?p & ~0))
(habitación (plazas-libres ?p & 1|2))
```

- Con **predicados o llamadas a funciones**. En este caso deben ir precedidos del signo **':'**

```
(habitacion (capacidad ?c) (plazas-libres ?p & : (> ?c ?p)))

(datos $?x & : (> (length$ $?x) 2))
```

- Predicados:
  - De tipo (todos terminan en **p**): ***numberp, floatp, symbolp, ...***
  - Comparaciones numéricas: **=, <>, <, <=, >, >=**
  - Igualdad (desigualdad) en **tipo y valor**: **eq (neq)**
  - Predicados definidos por el usuario.

Tipos de elementos condicionales (LHS)

- ***test*:** Se usa para evaluar expresiones en la parte izquierda de una regla, interviniendo en el proceso de ***pattern-matching***. El elemento condicional ***test*** se satisface si la llamada a la función que aparezca dentro de él **devuelve cualquier valor distinto de FALSE**

```
(defrule ejemplo-test
   (datos ?x)
   (valor ?y)
   (test (>= (abs (- ?y ?x)) 3))
=>
)
```

- ***or*:** Este EC se satisface cuando al menos uno de los componentes que aparece se satisface.

```
(defrule fallo-del-sistema
    (error-status desconocido)
    (or (temperatura alta)
          (valvula rota)
          (bomba (estado apagada))
    )
=>
    (printout t "El sistema ha fallado." crlf))
```

Tipos de elementos condicionales (LHS)

- ***and*:** CLIPS supone que todas las reglas tienen un ***and*** implícito que rodea todos los elementos condicionales de la LHS.

```
(defrule fallo-1
    (or (and (temperatura alta)(valvula cerrada))
        (and (temperatura baja)(valvula abierta))
    )
=>
   (printout t "El sistema tiene el fallo 1." crlf))
```

- ***not*:** Un elemento condicional negativo se satisface si **no existe ninguna entidad que cumpla** las restricciones expresadas.

```
(defrule libre
     (not (habitacion (plazas-libres ?p & ~0) (capacidad ?c & : (> ?c ?p)))
=>
)
```

- ***exists:*** Este EC permite que se produzca el ***pattern matching*** cuando **al menos exista un hecho que satisfaga la regla, sin tener en cuenta el número total de hechos que pudiesen *matchear***. Esto permite una sola activación para una regla con la que ***matcheen*** un conjunto de hechos.

Tipos de elementos condicionales

- ***forall*:** Permite el *matching* basado en un conjunto de EC's que son satisfechos por cada ocurrencia de otro EC. **Su funcionamiento es el contrario que el de *exists***. Para que el EC *forall* se satisfaga, **todo lo que *matchee* con <primer-EC> debe tener hechos que *matcheen* todos los demás EC que aparecen a continuación de <primer-EC>.**

```
(defrule todos-aprobados
  (forall
     (estudiante (nombre ?nombre) (ingles-aprobado ?nombre)
                                  (historia-aprobado ?nombre)
                                  (matematicas-aprobado ?nombre)
  )
  =>
  (printout t "Han aprobado todos los estudiantes." crlf))
```

- ← **:** Ligan la dirección de las entidades de la MT (hechos o instancias) que satisfacen el elemento condicional a una variable para poder realizar acciones sobre ellos.

```
(defrule habitacion-vacia
   ?vacía ← (habitación (capacidad ?c) (plazas-libres ?c) (numero ?n))
=>
   (printout t "Numero habitación vacía: " ?n crlf)
   (retract ?vacía))
```

Ejemplo: qué puede suceder?

```
(deffacts estudiantes "Todos los estudiantes iniciales"
      (estudiante (nombre Juan)(sexo varon)(fuma no)(alojado no))
      (estudiante (nombre Pepe)(sexo varon)(fuma no)(alojado no))
      (estudiante (nombre Luisa)(sexo mujer)(fuma no)(alojado no))
      (estudiante (nombre Pedro)(sexo varon)(fuma no)(alojado no))
)
```

```
(defrule pepe1
    ?x ← (estudiante (nombre Pepe)(sexo varon)(fuma no)(alojado no))
=>
     (printout t "se ha alojado pepe?" crlf)
     (bind ?aux (read))
     (modify ?x (alojado ?aux))
)
```

```
(defrule pepe2
    ?x ← (estudiante (nombre Pepe)(sexo ?)(fuma no)(alojado ?))
=>
     (printout t "se ha alojado pepe?" crlf)
     (bind ?aux (read))
     (modify ?x (alojado ?aux))
)
```

Propiedades de las reglas

- Las propiedades se declaran en la parte izquierda de una regla utilizando la palabra clave ***declare***. **Una regla sólo** puede tener **una sentencia *declare*** y debe aparecer antes del primer elemento condicional
- ***salience* :** Permite al usuario asignar una prioridad a una regla. La prioridad asignada debe estar en el rango -10000 .. +10000. Por defecto, la prioridad de una regla es 0.
  - Si la agenda incluye varias reglas, se desencadenará en primer lugar aquella que tenga **mayor prioridad**.

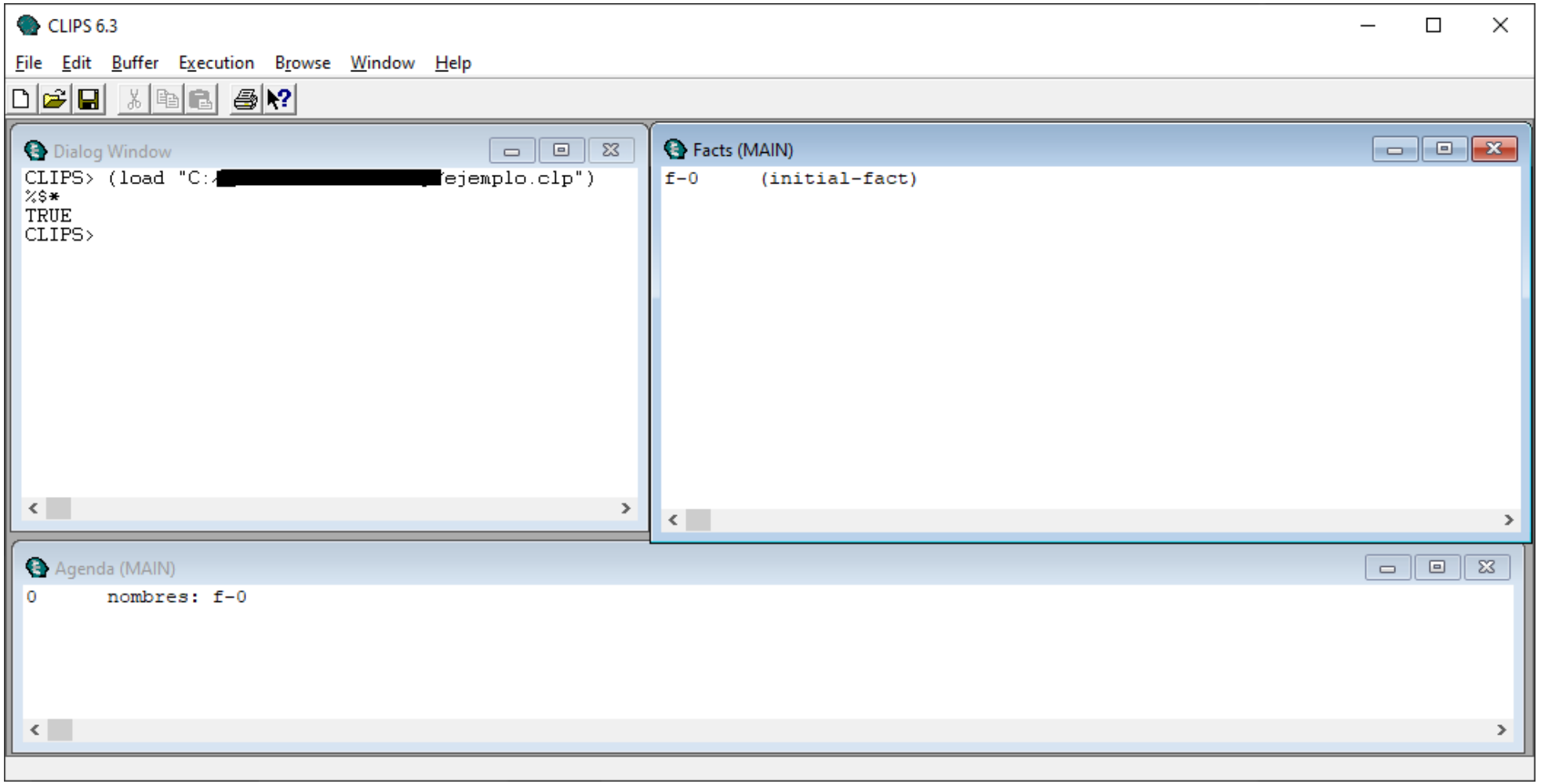
```
(deffacts coche (deposito vacio))

(defrule llenar-deposito
   (declare (salience 0))
   ?h ←  (deposito vacio)
=>
   (retract ?h)
   (assert (deposito lleno)))

(defrule iniciar-viaje
   (declare (salience 10))
   (deposito vacio)
=>
   (printout t "Me he quedado tirado!" crlf))
```

Atrapar los hechos

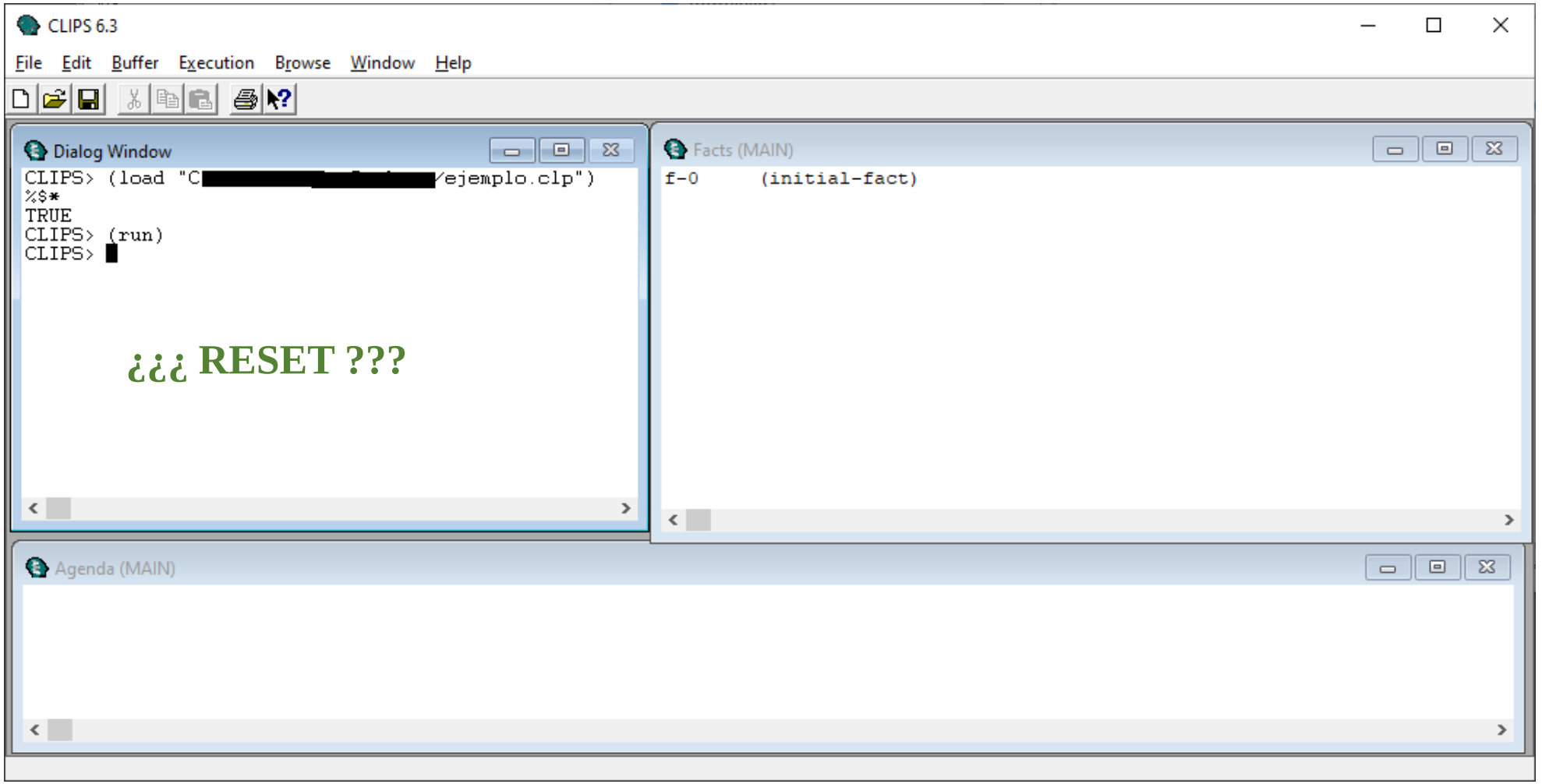
- Los hechos se pueden atrapar desde la condición de la regla, o ***do-for-fact / do-for-all-facts*:**

```
(deftemplate estudiante
    (multislot nombre)
    (slot sexo (allowed-values mujer varon))
    (slot fuma (allowed-values si no))
    (slot alojado (allowed-values si no)))
---------------------------------------------------------------------------------------------------------
(deffacts estudiantes "Todos los estudiantes iniciales"
     (estudiante (nombre Juan)(sexo varon)(fuma no)(alojado no))
     (estudiante (nombre Pepe)(sexo varon)(fuma si)(alojado no))
     (estudiante (nombre Luisa)(sexo mujer)(fuma no)(alojado no))
     (estudiante (nombre Pedro)(sexo varon)(fuma no)(alojado no)))
---------------------------------------------------------------------------------------------------------
(defrule nombres
(initial-fact)
=>
 (do-for-fact ((?var estudiante)) (eq ?var:fuma si)          (do-for-fact ((hecho1)(hecho*))(condición)
    (printout t "ID:" ?var "su nombre es:" ?var:nombre crlf)                  (operaciones))
;imprime el nombre SOLO del primer hecho que cumpla la condición  → (Pepe)
 ))
```

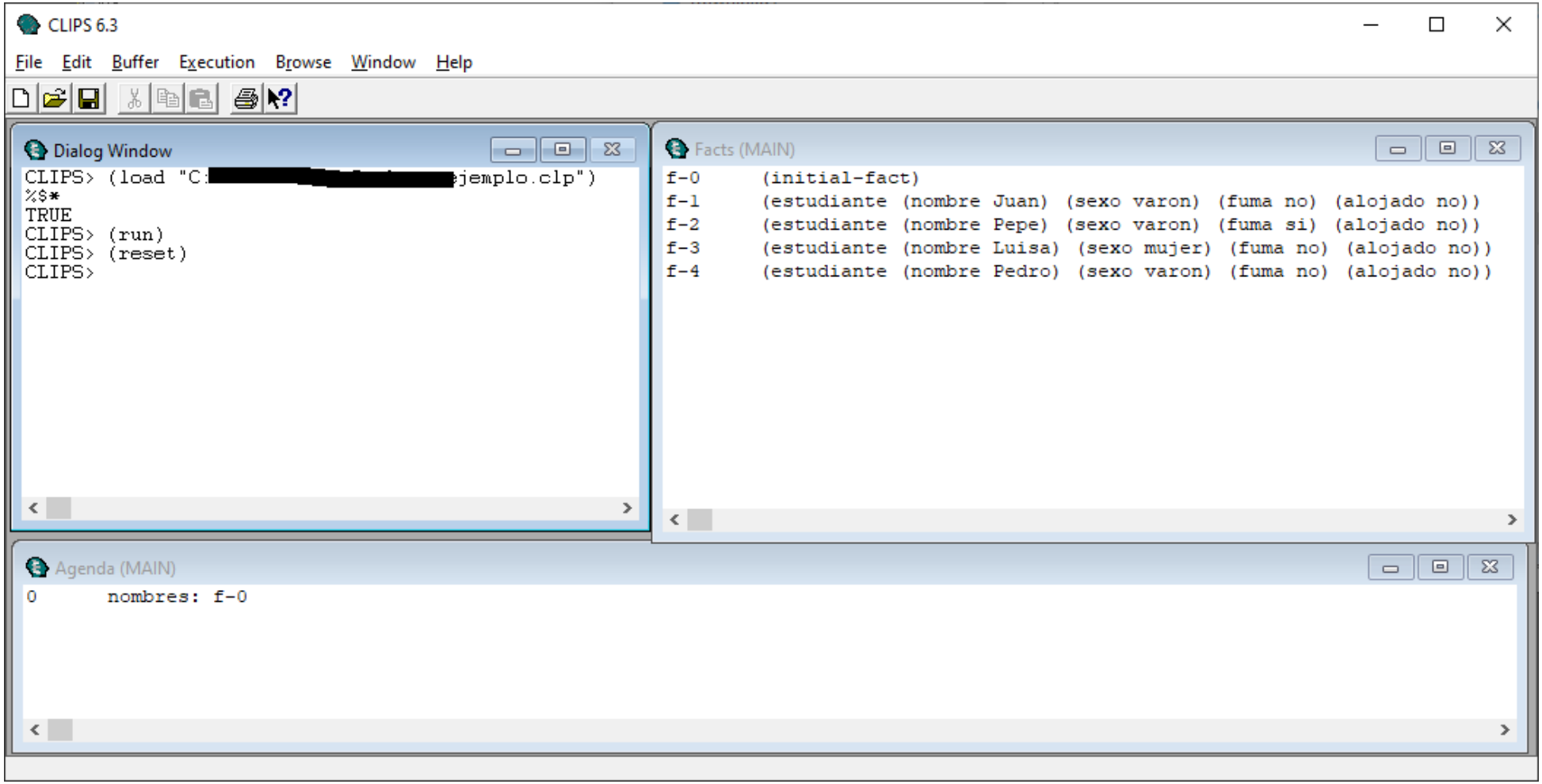
Atrapar los hechos



Atrapar los hechos

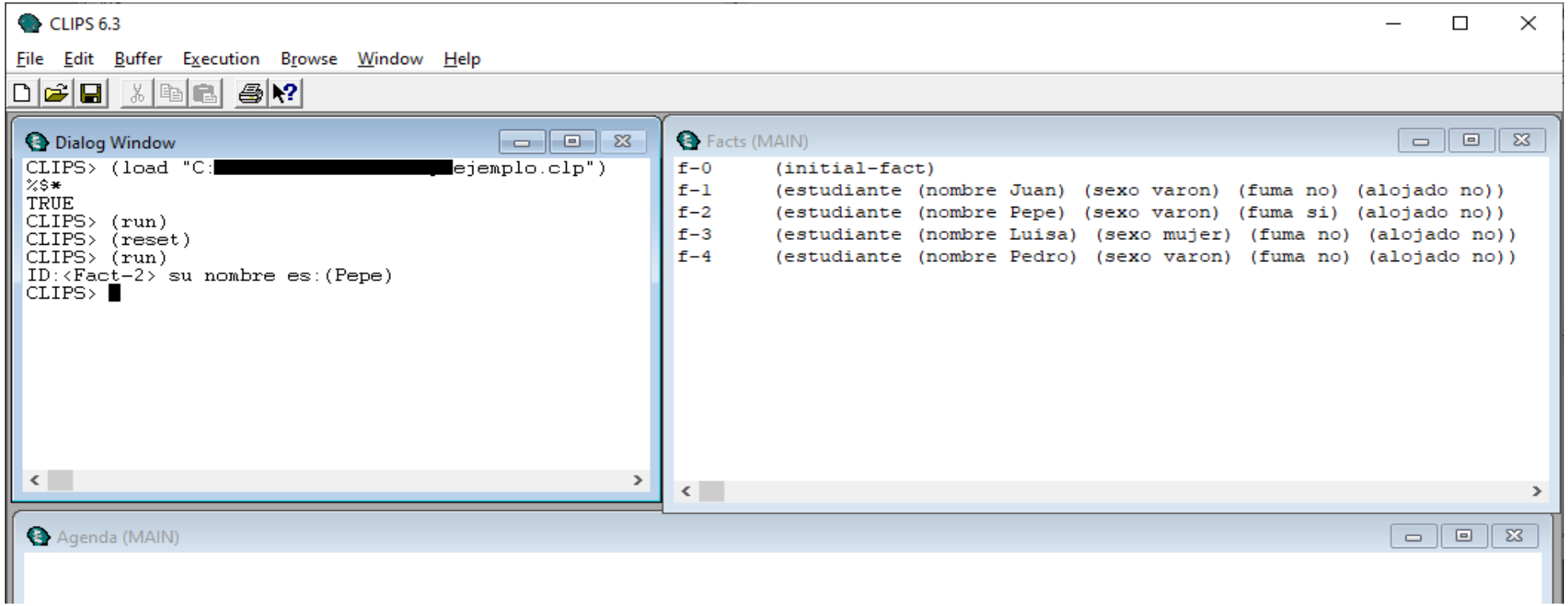


Atrapar los hechos



Atrapar los hechos

```
(defrule nombres
   (initial-fact)
=>
   (do-for-fact ((?var estudiante)) (eq ?var:fuma si)
      (printout t “ID:” ?var “su nombre es:” ?var:nombre crlf)
   )
)
```



Atrapar los hechos

```
(deftemplate estudiante
    (multislot nombre)
    (slot sexo (allowed-values mujer varon))
    (slot fuma (allowed-values si no))
    (slot alojado (allowed-values si no)))
---------------------------------------------------------------------------------------------------------
(deffacts estudiantes "Todos los estudiantes iniciales"
     (estudiante (nombre Juan)(sexo varon)(fuma no)(alojado no))
     (estudiante (nombre Pepe)(sexo varon)(fuma no)(alojado si))
     (estudiante (nombre Luisa)(sexo mujer)(fuma no)(alojado si))
     (estudiante (nombre Pepa)(sexo mujer)(fuma no)(alojado si)))
---------------------------------------------------------------------------------------------------------
(defrule nombres
(initial-fact)
=>
 (do-for-all-facts ((?var estudiante)) (and(eq ?var:alojado si)(eq ?var:sexo varon))
     (printout t “ID: “ ?var “ su nombre es:" ?var:nombre crlf) ;imprime TODOS los nombres de los varones que están alojados
 )
)
```

Atrapar los hechos desde funciones

```
(deftemplate estudiante
    (multislot nombre)
    (slot sexo (allowed-values mujer varon))
    (slot fuma (allowed-values si no))
    (slot alojado (allowed-values si no)))
```

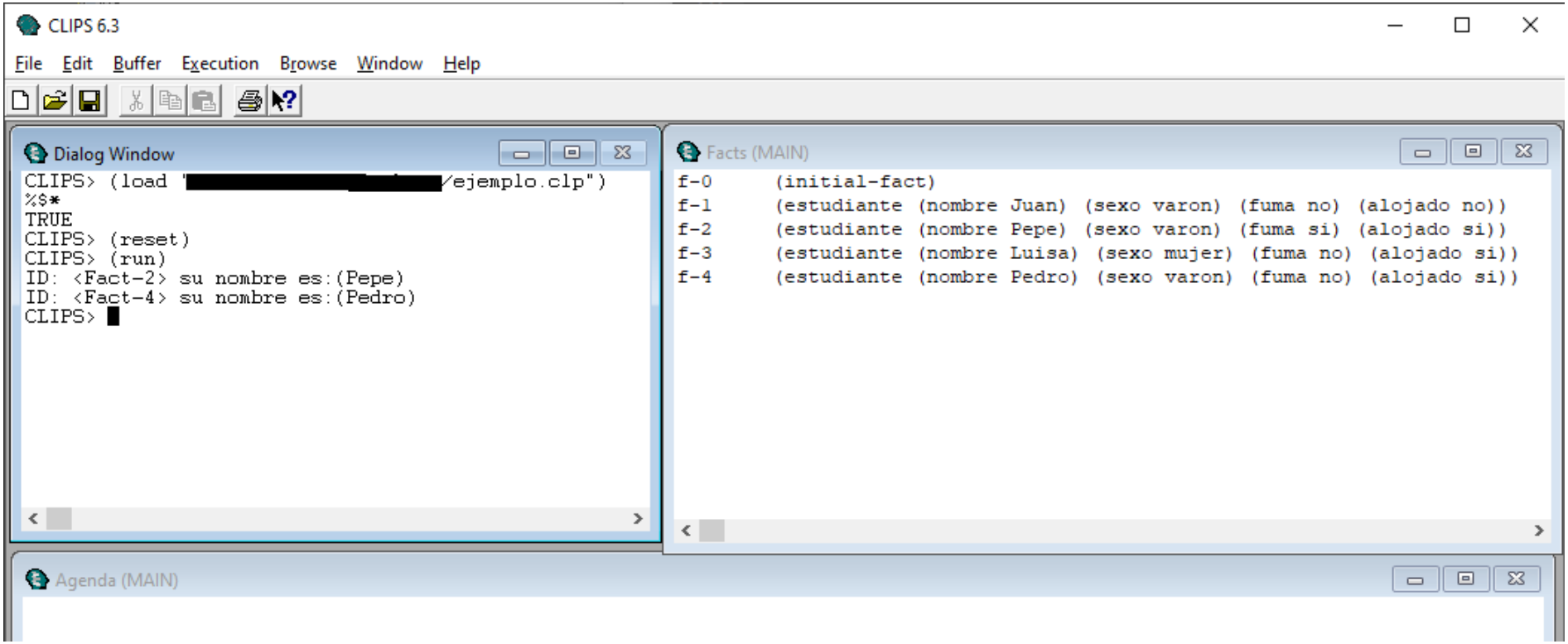
```
(deffacts estudiantes "Todos los estudiantes iniciales"
     (estudiante (nombre Juan)(sexo varon)(fuma no)(alojado no))
     (estudiante (nombre Pepe)(sexo varon)(fuma no)(alojado si))
     (estudiante (nombre Luisa)(sexo mujer)(fuma no)(alojado si))
     (estudiante (nombre Pepa)(sexo mujer)(fuma no)(alojado si)))
```

```
(deffunction funcion ()
   (do-for-all-facts ((?var estudiante)) (and(eq ?var:alojado si)(eq ?var:sexo varon))
       (printout t "ID: " ?var " su nombre es:" ?var:nombre crlf)
   )
)
```

```
(defrule nombres
(initial-fact) ; sin este hecho también atraparía initial-fact
=>
 (funcion)
)
```

Atrapar los hechos

```
(defrule nombres
(initial-fact)
=>
  (do-for-all-facts ((?var estudiante)) (and(eq ?var:alojado si)(eq ?var:sexo varon))
      (printout t “ID: “ ?var “ su nombre es:" ?var:nombre crlf)
  )
)
```



Atrapar los hechos

```
(deftemplate estudiante
    (multislot nombre)
    (slot sexo (allowed-values mujer varon))
    (slot fuma (allowed-values si no))
    (slot alojado (allowed-values si no)))
```

```
(deftemplate padre
    (multislot nombre)
    (slot alojado (allowed-values si no))))
```

```
(deffacts personas "Todos los estudiantes iniciales"
     (estudiante (nombre Juan)(sexo varon)(fuma no)(alojado no))
     (estudiante (nombre Pepe)(sexo varon)(fuma no)(alojado si))
     (estudiante (nombre Luisa)(sexo mujer)(fuma no)(alojado si))
     (estudiante (nombre Pedro)(sexo varon)(fuma no)(alojado si)))
     (padre (nombre Pedro) (alojado si))
     (padre (nombre Manuel) (alojado si))
```

```
(defrule nombres
(initial-fact)
=>
 (do-for-all-facts ((?var1 estudiante)(?var2 padre)) (eq ?var1:nombre ?var2:nombre))
     (printout t "su nombre es:" ?var1:nombre crlf) ;imprime el nombre de los estudiantes que existan padres con el mismo nombre
 )
)
```

Ejemplo

Lista de hechos

f-0 (initial-facts)
f-1 (bloque rojo)
f-2 (bloque verde)
f-3 (bloque verde encima-de rojo)
f-4 (camion (bloques))

---

f-0 (initial-facts)
f-1 (bloque rojo)
f-2 (bloque verde)
f-4 (camión (bloques))
f-5 (bloque verde no tiene un bloque encima)
f-6 (bloque rojo no tiene un bloque encima)

---

f-0 (initial-facts)
f-2 (bloque verde)
f-5 (bloque verde no tiene un bloque encima)
f-7 (camión (bloques rojo))

Lista de reglas

```
(defrule grua
  ?b ← (bloque verde encima-de rojo)
=>
   (retract ?b)
   (assert (bloque verde no tiene un bloque encima))
   (assert (bloque rojo no tiene un bloque encima)))
```

```
(defrule grua-camnion
  ?b ← (bloque rojo no tiene un bloque encima)
  ?c ← (camion (bloques $?bl))
  ?r ← (bloque rojo)
=>
   (retract ?b ?r)
   (bind $?bl (insert$ $?bl 1 rojo))
   (modify ?c (bloques $?bl)) )
```

Ejercicio 1

Crea un sistema para simular el estado de una persona.

Los únicos posibles estados son, durmiendo, comiendo o estudiando (por defecto será durmiendo).

El sistema preguntará el nuevo estado, se introducirá por teclado y cambiará el estado de la persona.

Si no se introduce un estado válido (arriba mencionados), no se modificará el estado de la persona.

Ejercicio 2

Crea un sistema para simular un restaurante donde solamente se dan postres. Como postre se puede escoger fruta o dulce.

fruta: plátano, manzana o kiwi

dulce: tarta de queso, tarta de manzana o coulant

Solamente hay X unidades de productos. Cuando se acaben las unidades, ese postre desaparecerá del menú.

Al principio el restaurante está vacío. En la puerta existe una cámara que reconoce los clientes, generando el hecho automáticamente:

(nuevo cliente ***nombre_cliente***)

De la misma manera, cuando el cliente salga de la puerta se generará el hecho:

(adiós ***nombre_cliente***) para poder eliminar el cliente

Cuando un cliente le diga lo que quiere al camarero, éste generará el hecho:

(quiere **nombre_cliente nombre_postre**)

Ejercicio 3

Se le echa de menos a Pacman y por eso, vamos a simular el juego de Pacman.

El mundo de Pacman es una matriz de 4x4. Pacman, el fantasma y la única comida de Pacman se colocarán aleatoriamente en una casilla.

El movimiento de Pacman lo realizaremos insertando un hecho (mover arriba) o (mover abajo) o (mover izquierda) o (mover derecha).

El movimiento del fantasma se realizará aleatoriamente.

Los movimientos se harán por turnos, empezando Pacman.

El juego terminará cuando Pacman coma la comida o el fantasma y Pacman se encuentren en la misma casilla.