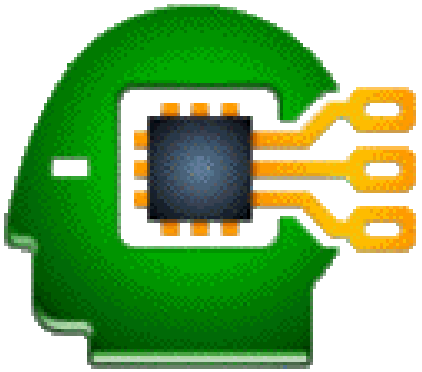


Inteligencia Artificial

Representación del conocimiento y razonamiento

ROBOTIKA & SISTEMA AUTONOMOEN IKERKETA TALDEA

eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea

Sistemas expertos

- Al rededor de los **años 70’s** se empezaron **a crear sistemas expertos**.

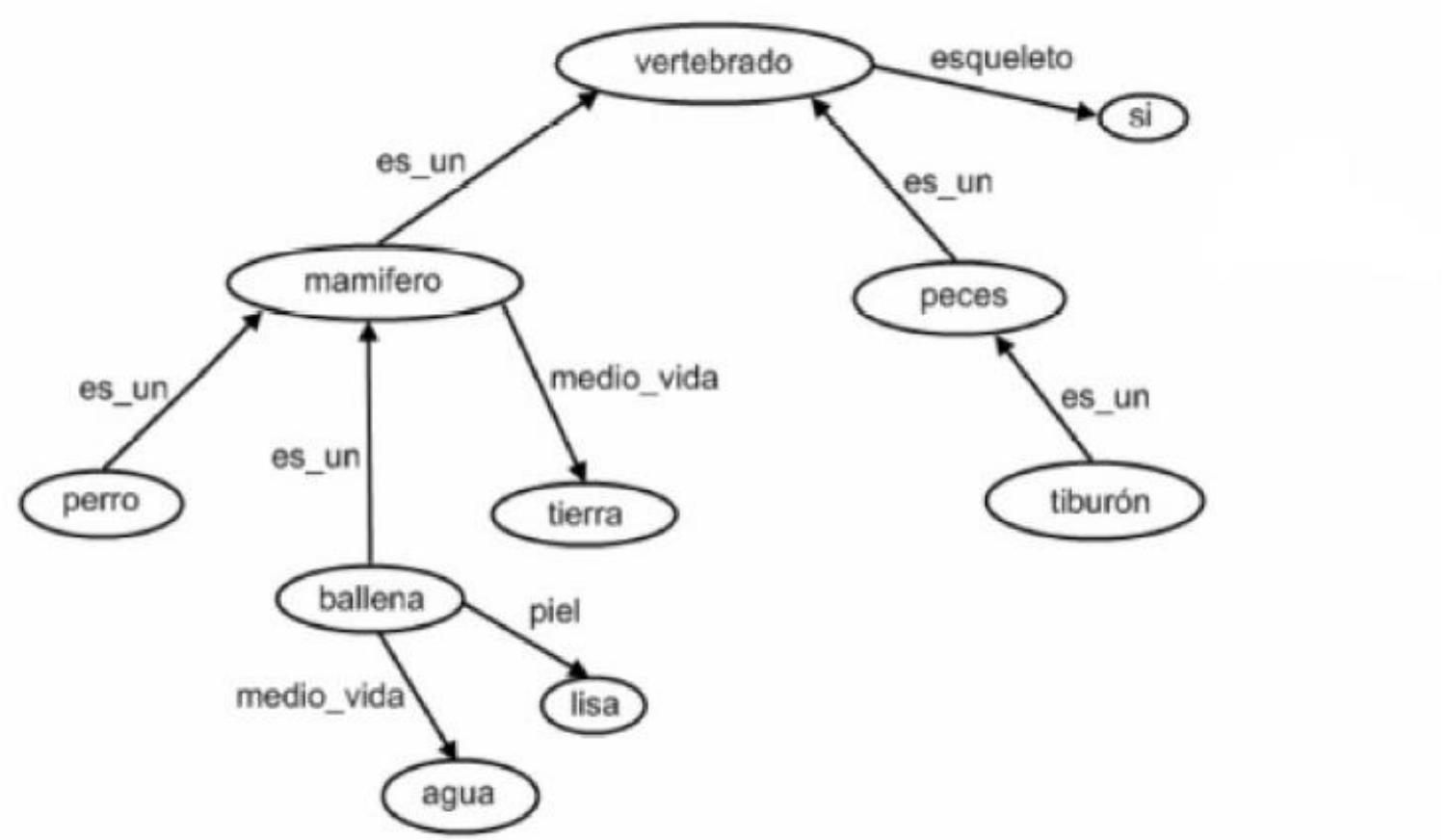
- Estos “Sistemas Expertos” se llaman así por la pretensión de **simular el comportamiento de un humano experto,** en un área muy técnica y específica. También conocidos como **Sistemas Basados en el Conocimiento** (SBC).

- Todo problema es más **sencillo de resolver si disponemos de conocimiento** específico sobre él.

- Para poder operar con el conocimiento **es necesario representarlo**.

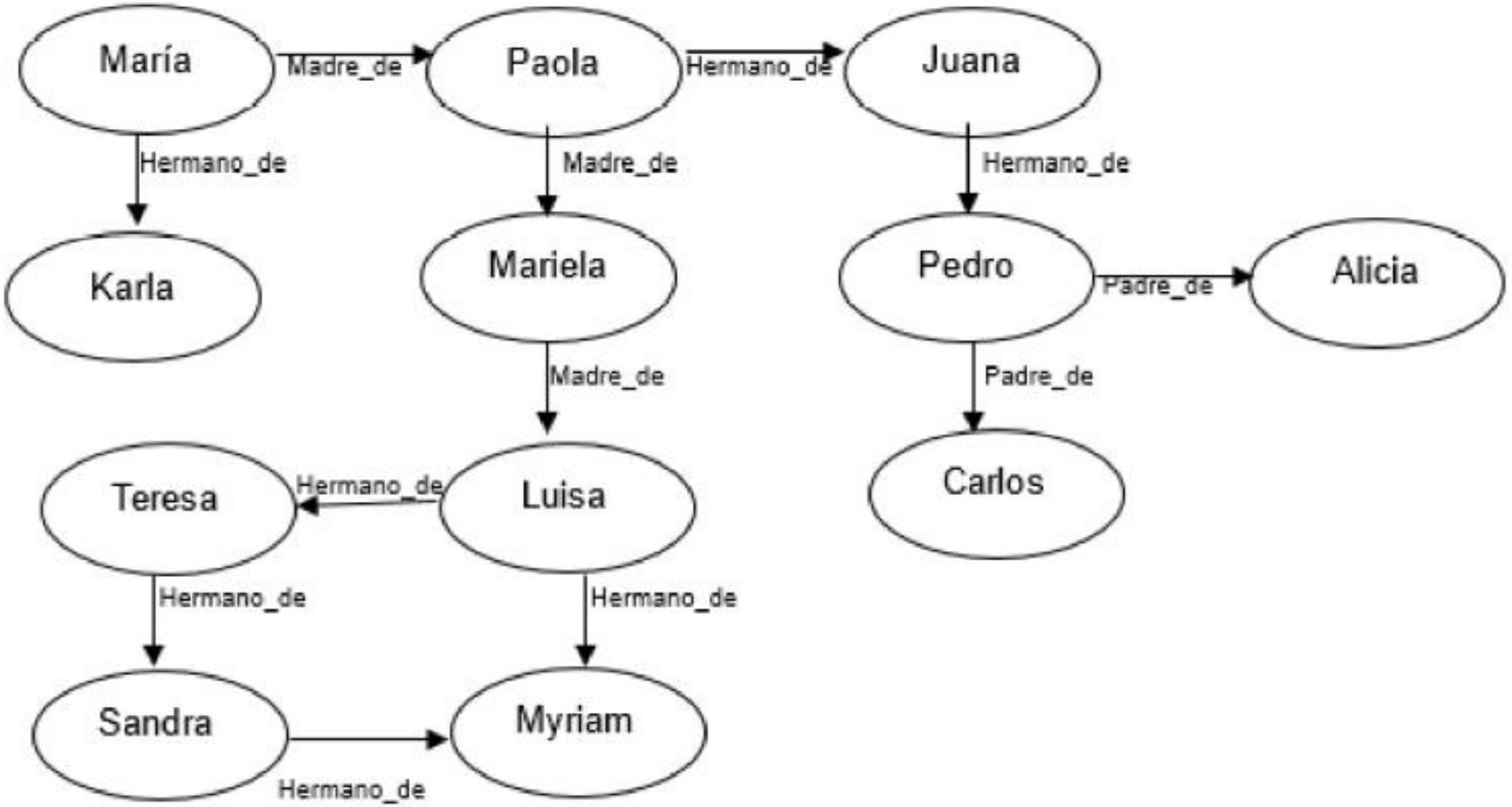
Representación (redes semánticas)

- **Redes semánticas**: Es la representación del conocimiento **mediante nodos**, que son los elementos del **conocimiento o los conceptos** y por ramas o **arcos**, que son **las relaciones entre los nodos**.



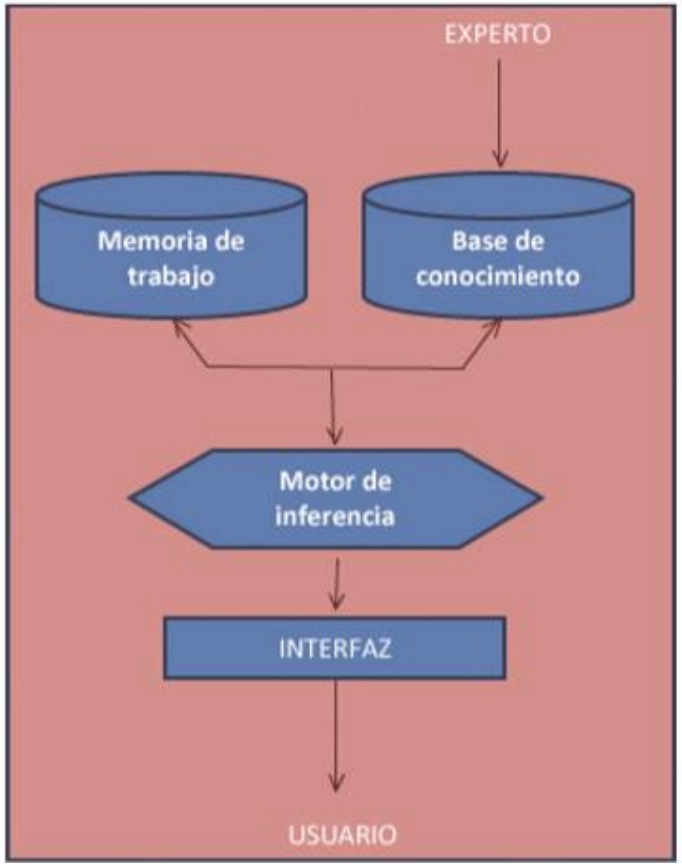
Representación (mapas conceptuales)

- **Mapas conceptuales**: Es la representación del conocimiento mediante **una red semántica utilizando la lógica de predicados**.

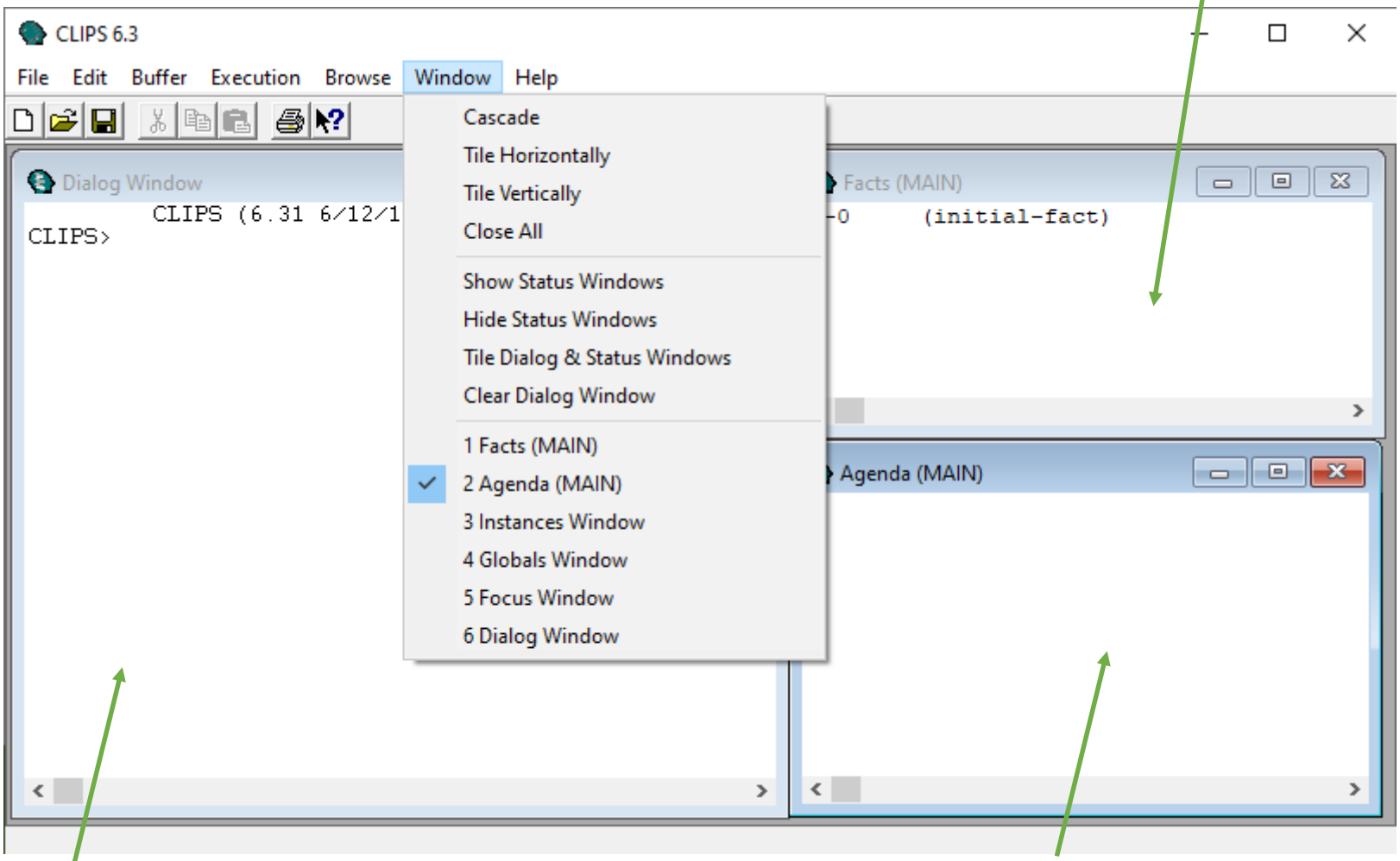


Representación y razonamiento

- La mayor parte de los **SBC** están constituidos fundamentalmente **por tres componentes**:
  - **Memoria de trabajo**: almacena información sobre el problema (**representación**).
  - **Base de conocimiento**: almacena el conocimiento sobre el problema (**acciones**).
  - **Motor de inferencia**: contiene información relativa al funcionamiento del sistema.



Representación y razonamiento



Memoria de Trabajo (representación)

- También llamada **base de hechos (BH)**.
- Contiene ***hechos*** sobre el mundo (**representación del estado**), que pueden observarse directamente o deducirse a través de las reglas**.**
  - **Se definen mediante variables simples o multicampo**
- ***Ejemplos:***
  - *"El bloque rojo está situado encima del bloque verde"*
  - *(a b c d e)*
  - *(tanque "vacío")*

- **Los hechos** se pueden **modificar** mediante reglas, incluso **añadir** o **eliminar**

Base de Conocimiento (acciones)

- También llamada **Base de reglas (BR).**
- **Cada regla** representa **un paso de la resolución del problema**. Se utilizan para representar las ***acciones del problema***.
- Una regla **es un conjunto de condiciones junto con las operaciones a realizar si las condiciones se satisfacen.**
- Las reglas son conocimiento duradero, persistente, sobre el dominio. Una vez definidas, no se modifican, solo el experto del dominio puede modificarlas.
- ***Ejemplos****: simulación de una grúa que transporta boques*

REGLA: *sujetar bloque rojo*
(bloque rojo está encima del bloque verde) **y**
(no hay un bloque encima del bloque rojo)
=>
(bloque rojo sujeto por la grúa)
(bloque rojo no está encima del bloque verde)
(bloque verde no tiene un bloque encima)

Motor de inferencia

- Procesa la información de la **BH y BR**.
- **Es el mecanismo de razonamiento**.
- Selecciona una regla de la BR y la aplica.

CLIPS: hechos y reglas

- **Las reglas** son como **sentencias IF-THEN** de lenguajes procedurales como C, Ada, Java ...

- CLIPS mantiene una **lista de hechos** y **reglas**, permitiendo éstas **operar con los hechos almacenados** en la lista de hechos, dado que **los hechos son necesarios para disparar o activar las reglas.**

Lista de hechos (representación**)**	**Lista de reglas (**acciones**)**
(tiempo llueve) (tiempo viento) (tiempo 10 grados) (persona sin_paraguas)	(defrule ***nombre_regla*** (tiempo llueve) => (printout t “coger paraguas” crlf))

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.**

(color azul) **(padre_de Juan Pepe)** **(nombre "Juan Manuel")** **(pacman 3 8 2 3 30)**

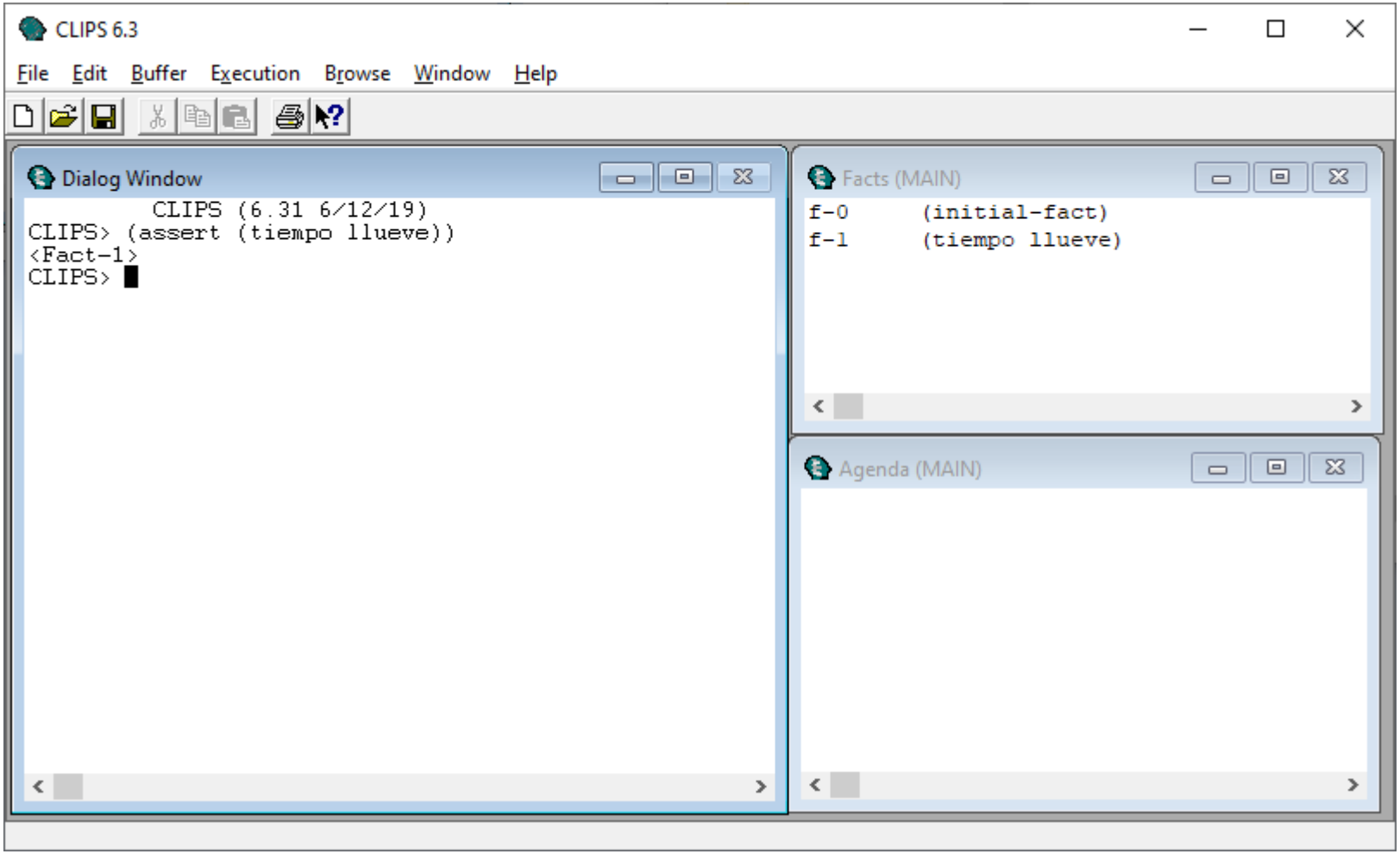
Hechos: operaciones

- Las acciones que se pueden realizar sobre los hechos:
  - Acciones que modifican la MT: lista de hechos
    - para **insertar** hechos: ***(assert <hecho>+)***
    - Para **borrar** hechos: ***(retract <especificador-hecho>)*** → **Nota1**
  - 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

Insertando hechos

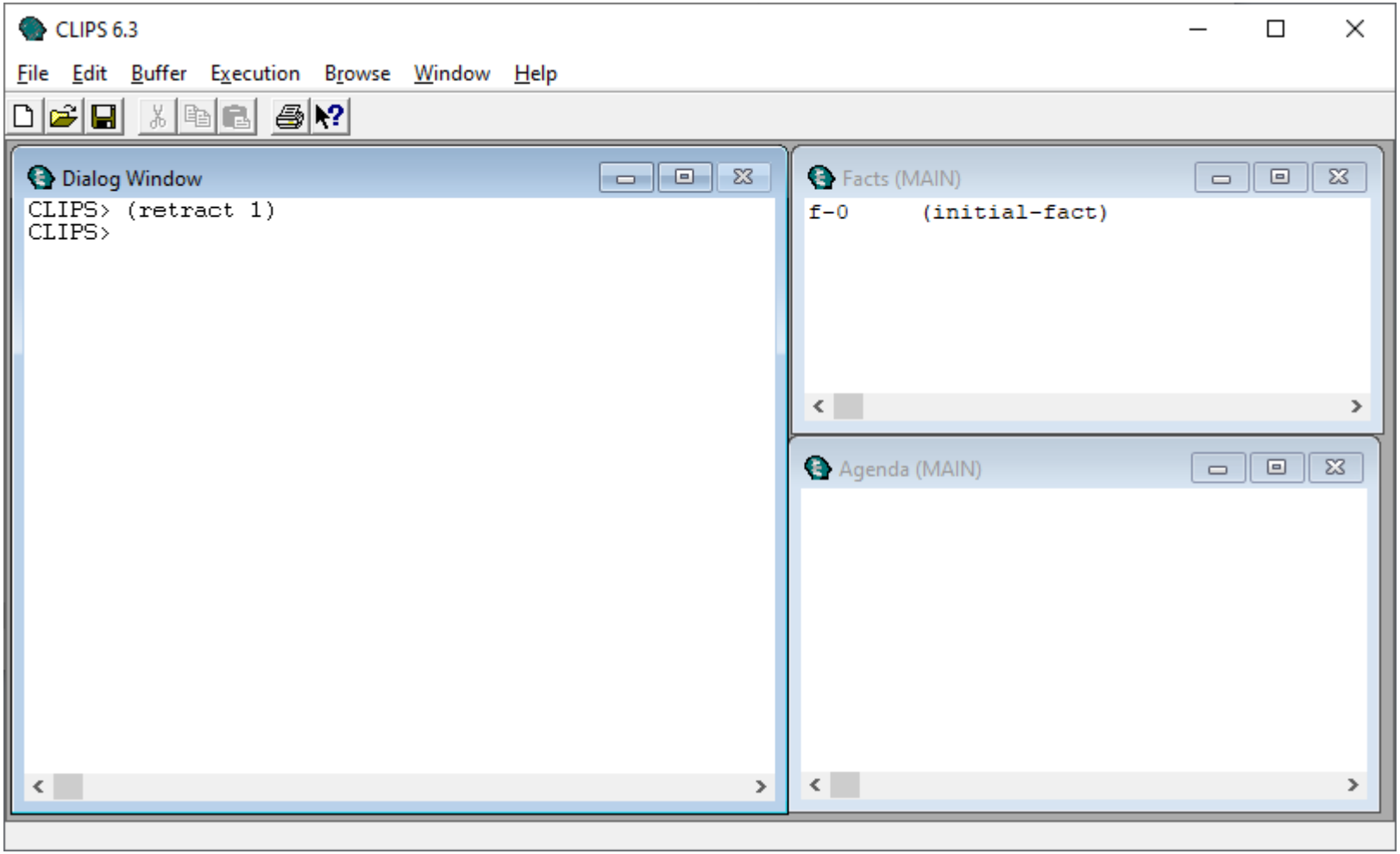
- El comando **(assert)** añade un hecho. Cada hecho se identificará a continuación mediante **un índice único** y será introducido en la memoria de trabajo (MT).



- El hecho ***initial-fact*** está por defecto, y puede ser usado para iniciar nuestro sistema

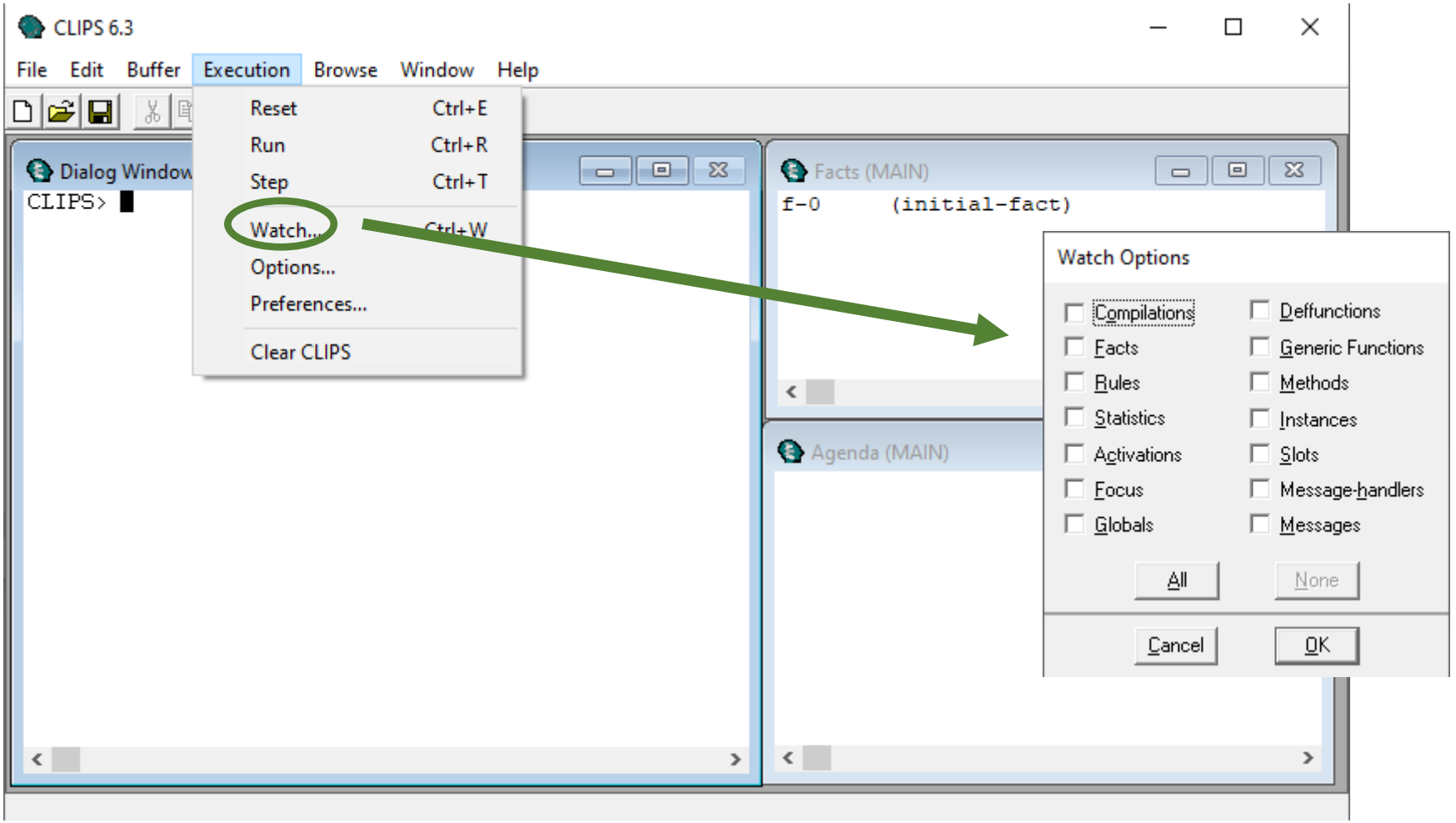
Eliminando hechos

- **Para eliminar un hecho** (**o varios**), se usa el comando **(retract)** con un identificador:
  - **(retract 1)** elimina el hecho con **identificador 1** **(como saber el identificador? ver 24)**
  - **(retract *)** elimina todos los hecho



Más información

- **Comandos:** ***(watch facts | rules | acticationes | methods | deffunctions | global | slots | all)*** Como **herramienta de depuración** se puede utilizar la orden ***watch***
- **Desde interfaz:**



Comandos: hechos

- Tecleando **(facts)** nos permite **ver la lista completa de hechos actual**, apareciendo cada hecho junto a su identificador único.

```
CLIPS> (facts)
f-0     (initial-fact)
f-1     (tiempo llueve)
For a total of 2 facts
```

- Para listar los hechos **a partir de un identificador** usamos **(facts <ident>)**
- Para mostrar **un rango concreto de hechos:** **(facts <idmin> <idmax>)**

```
CLIPS> (reset)
CLIPS> (assert (bola azul))
<Fact-1>
CLIPS> (assert (bola roja))
<Fact-2>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (bola azul)
f-2     (bola roja)
For a total of 3 facts.
```

```
CLIPS> (retract 1)
CLIPS> (facts)
f-0     (initial-fact)
f-2     (bola roja)
For a total of 2 facts.
```

Hechos: definición de hechos iniciales

- En un problema, “siempre” habrá una representación inicial: mundo pacman, etc
- El constructor ***deffacts*** permite especificar un conjunto de hechos como conocimiento inicial.

(deffacts <nombre-colección-hechos> [<comentario>] <patrón-RHS>*)

```
(deffacts arranque "Estado inicial del frigorífico"
    (frigorífico interruptor encendido)
    (frigorífico puerta abierta)
    (frigorífico temperatura (get-temp))) ; get-temp → función que se ha implementado
```

- Se puede insertar la representación inicial mediante **assert**
- Con el comando ***reset*** se eliminan todos los hechos que hubiera en la lista de hechos actual, y a continuación añade los hechos correspondientes a sentencias ***deffacts***. Siempre es recomendable y/o necesario al inicio ejecutar el comando ***reset*****.**
- El comando **(clear)** elimina todos los hechos y reglas.

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
)
```

Reglas

- **Una regla** CLIPS es una entidad **independiente**: no es posible **el paso de datos** entre dos reglas. Sintaxis:

```
defrule <nombre-regla> [“comentario”]
   <elemento-condición>* ; Parte izquierda (LHS)
=>
   <acción>*) ; Parte derecha (RHS) de la regla
```

```
(defrule regla-ejemplo
   (frigorífico interruptor encendido)
   (frigorífico puerta abierta)
=>
   (assert (frigorífico comida estropeada)))
```

- **Si se introduce** en la base de reglas **una nueva regla con el mismo nombre** que el de una existente, **la nueva regla reemplazará a la antigua**.
- El “comentario”, un ***string***, se usa normalmente para describir el propósito de la regla.
- Si una regla **no tiene parte izquierda**, entonces el hecho ***(initial-fact)*** actuará como el elemento condicional, y **la regla se activará cada vez que se ejecute el comando *reset***.
- También puede no haber ninguna acción en el consecuente, con lo que la ejecución de la regla no tiene en ese caso ninguna consecuencia.

Como se activa/dispara una regla?

- ¿Cómo se satisfacen los EC de una regla? **La satisfactibilidad de un EC se basa en la existencia o no existencia en la MT (representación) de los hechos especificados**

Memoria de trabajo: hechos

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

- Para ***matchear*** con una regla sus campos deben aparecer en el mismo orden que indique la regla. Es decir, **los hechos ordenados "codifican" la información según la posición**.

```
(defrule regla
   (padre_de Juan Pepe)
=>
   (printout t "El padre de Pepe es Juan" crlf)
)
```

```
(defrule regla
   (padre_de ?p ?h)
=>
   (printout t "El padre de " ?h " es " ?p crlf)
)
```

Como se activa/dispara una regla?

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

Memoria de trabajo: hechos

```
(pacman 3 8 2 3 30)
```

- ***Restricción literal:***

```
(defrule regla
   (pacman 3 8 2 3 30)
=>
   (printout t "OK" crlf)
)
```

- ***Con comodines simples y multicampo :***

```
(defrule regla
   (pacman $?)
=>
   (printout t "OK" crlf)
)
```

- ***Con variables simples y multicampo:***

```
(defrule regla
   (pacman $?p)
=>
   (printout t "El estado es " $?p crlf)
)
```

```
(defrule regla
   (pacman 3 ? 2 $?resto)
=>
   (printout t "El resto es " $?resto crlf)
)
```

Elementos Condicionales (EC)

- **EC:** especifica restricciones sobre elementos de las listas de ***hechos***: sólo se satisface si existe una entidad que cumple las restricciones expresadas.
- **La condición** o antecedente de una regla **puede ser múltiple**:

```
(defrule cielo
   (color azul)
   (estamos-en exterior)
=>
   (assert (cielo-es despejado)))
```

- Al igual que la acción resultante, p.e. para mostrar un mensaje por la salida estándar con ***printout, llamada a una función …***

```
(defrule cielo
   (color azul)
   (estamos-en exterior)
=>
   (assert (cielo-es despejado))
   (vamos_a_la_playa) )
```

función

- **Una regla se ejecuta cuando**:
  - **todos sus elementos condicionales son satisfechos** por la lista de hechos y/o la lista de instancias (en caso de clases).
  - **el motor de inferencia la selecciona**.

Tipos de elementos condicionales (LHS)

- ***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)
       (válvula rota)
       (bomba apagada)
   )
=>
   (printout t "El sistema ha fallado." crlf))
```

- ***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)(válvula cerrada))
        (and (temperatura baja)(válvula abierta))
    )
=>
   (printout t "El sistema tiene el fallo 1." crlf))
```

Tipos de elementos condicionales

- ← **:** 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 0 0 0 0)
=>
   (printout t “Habitación vacía" crlf)
   (retract ?vacía))
```

```
(defrule habitacion-vacia
   ?vacía ← (habitación $?x)
=>
   (bind ?i 1)
   (bind ?seguir true)
   (while (and(<= ?i (length$ $?x))(eq ?seguir true))
      (if (not(= (nth$ ?i $?x) 0)) then
         (bind ?seguir false)
      )
      (bind ?i (+ ?i 1))
   )
   (if (eq ?seguir true) then
      (printout t “Habitación vacía: " ?n crlf)
      (retract ?vacía)
   ))
```

Comandos: reglas

- Una vez introducida una o varias reglas, tecleando ***(rules)*** aparecen los nombres de las presentes en la base de conocimiento. El contenido de una regla con sus comentarios se muestra con ***(ppdefrule <nombre-regla>)*** y ***(undefrule <nombre-regla>)*** la elimina (***** las elimina todas):

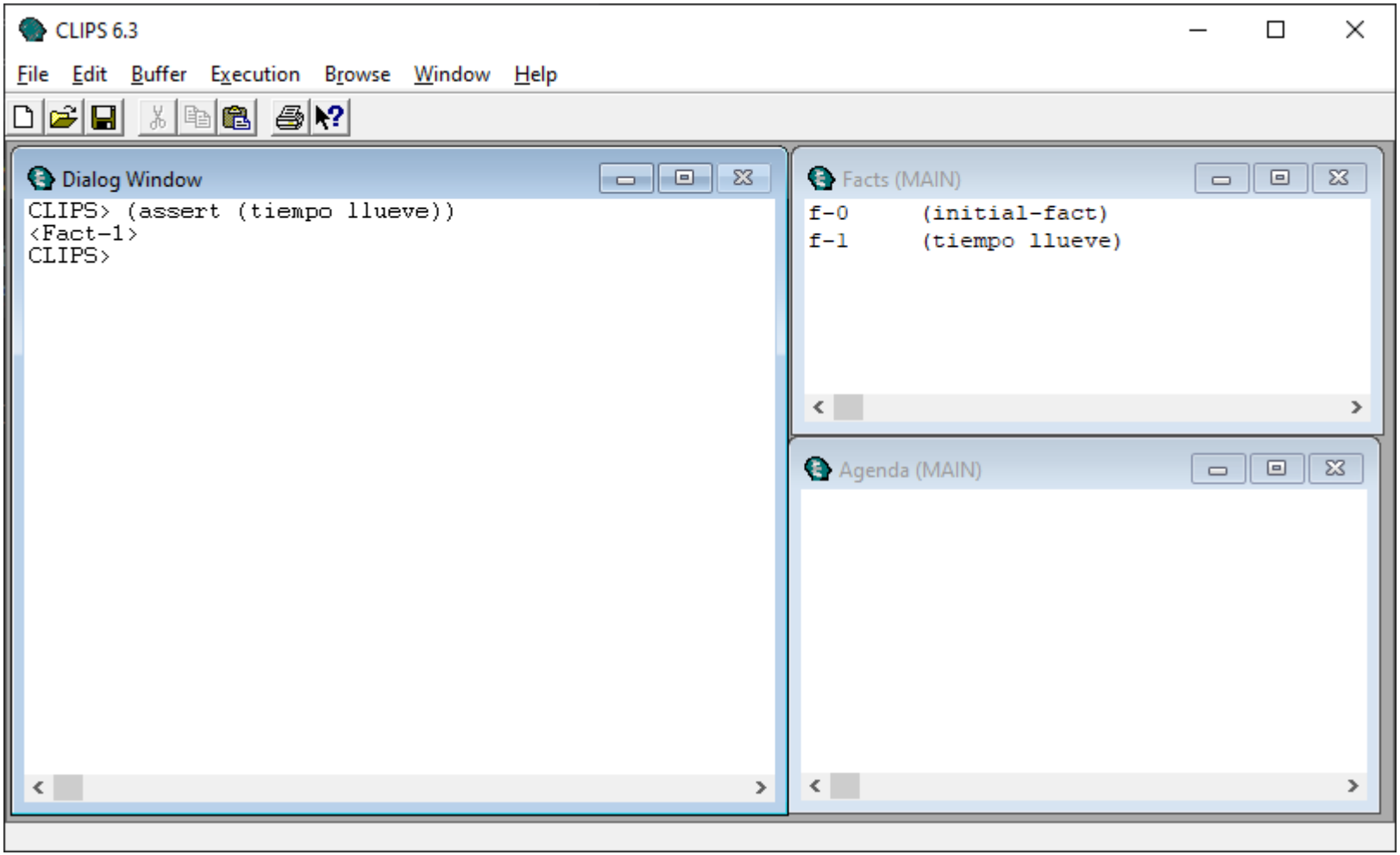
```
CLIPS> (defrule r1 (tiempo nublado) => (assert (coger paraguas)))
CLIPS> (rules)
r1
For a total of 1 defrule.

CLIPS> (ppdefrule r1)
(defrule MAIN::r1
   (tiempo nublado)
   =>
   (assert (llevar paraguas)))

CLIPS> (undefrule r1)
CLIPS> (rules)
CLIPS>
```

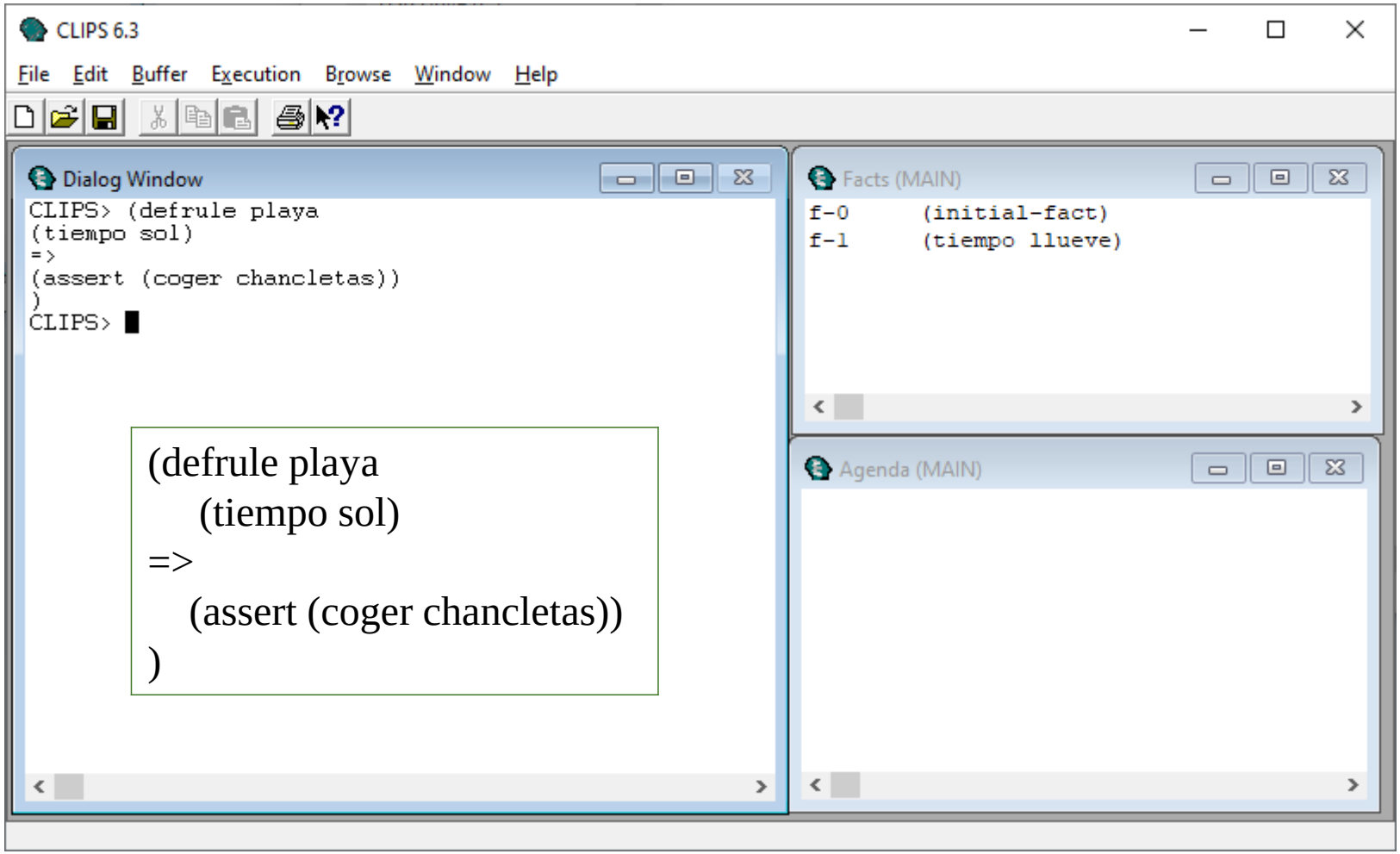
CLIPS

- Introducimos **(tiempo llueve)** en la MT (representación)



CLIPS

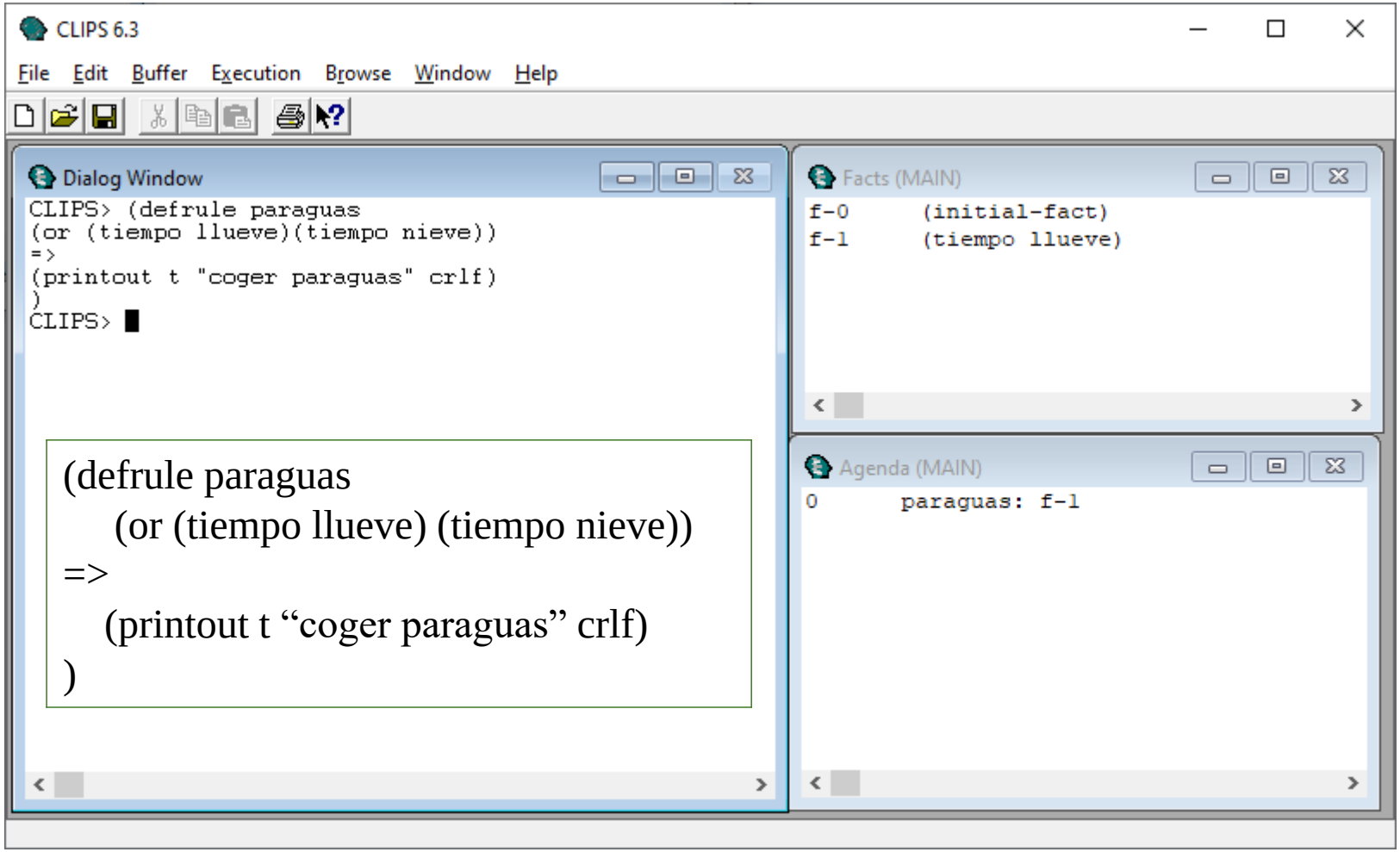
- Introducimos una regla llamada ***playa*** en la Base de Conocimiento



- Por qué no aparece en “Agenda?”

CLIPS

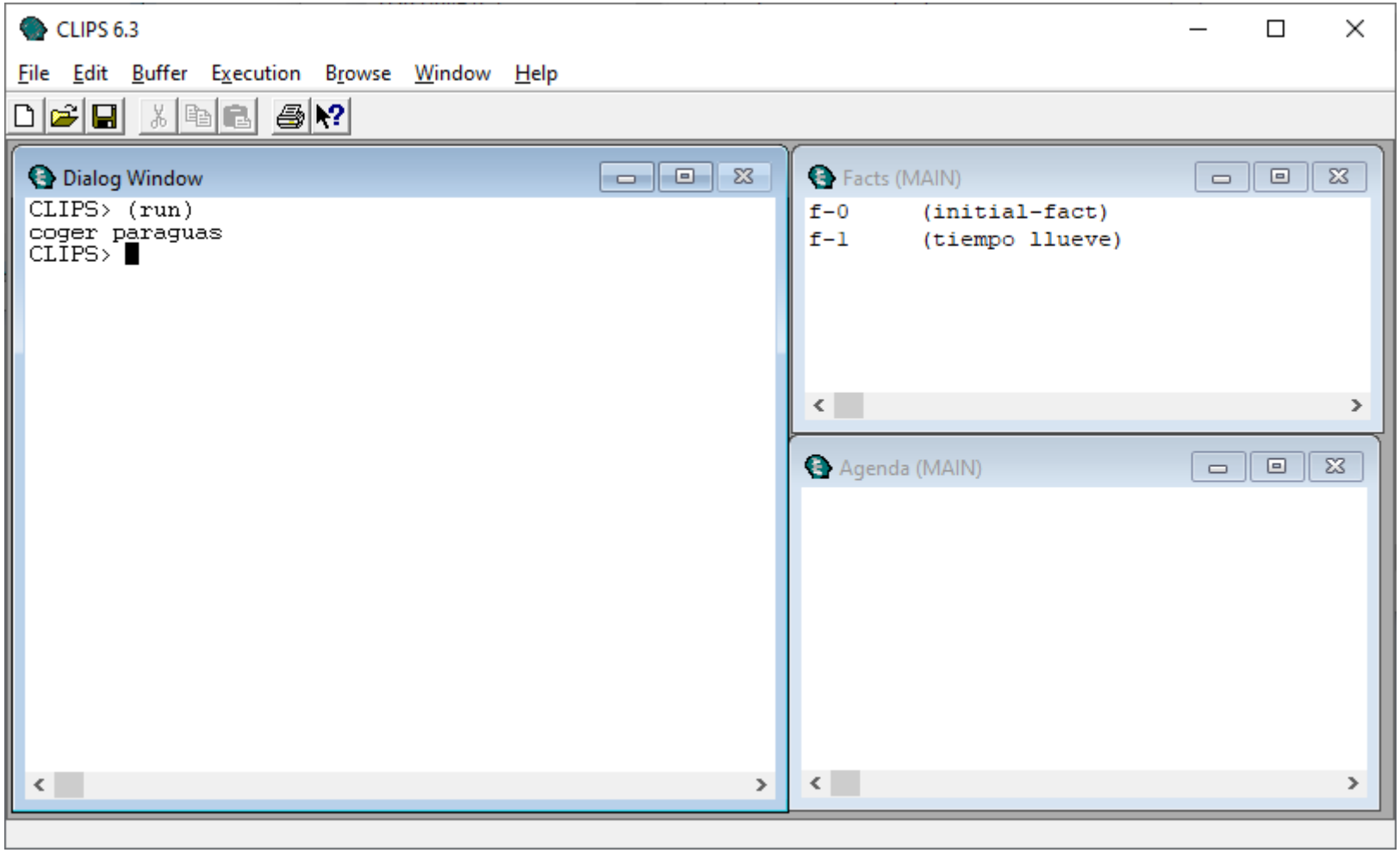
- Introducimos una regla llamada ***paraguas*** en la Base de Conocimiento




```
(defrule paraguas
    (or (tiempo llueve) (tiempo nieve))
=>
   (printout t “coger paraguas” crlf)
)
```

CLIPS

- Ejecutamos “el programa” con el comando **(run)**. También es posible **(run 5)** por ejemplo



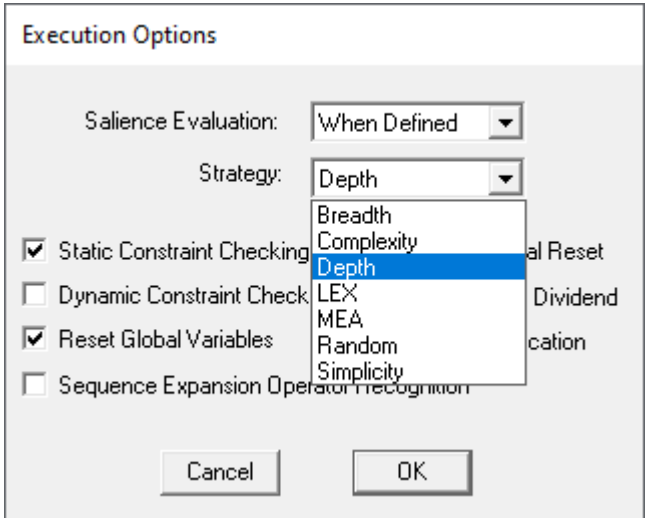
- Por qué el programa no sigue?
  - Si no se modifica la BD, una regla sólo se ejecutará una única vez!

Motor de inferencia

- El motor de inferencia **intenta emparejar o encajar los hechos de la lista de hechos con los patrones de las reglas**.

- Si el encaje **ocurre para todos los patrones de una regla**, ésta se **activa** o se **dispara**.
  - Cuantas veces se ejecuta una regla?
    - Esta comprobación para una regla **sólo se realiza tras eliminar un hecho y volver a añadir, o al modificar (que tengan que ver con la regla)**

- La **agenda** (base de conocimiento BC) almacena la lista de reglas activadas **siguiendo un orden de prioridad.**
  - Existen **distintas estrategias** de resolución de conflictos que permiten establecer criterios a la hora de insertar una activación de regla en la agenda.



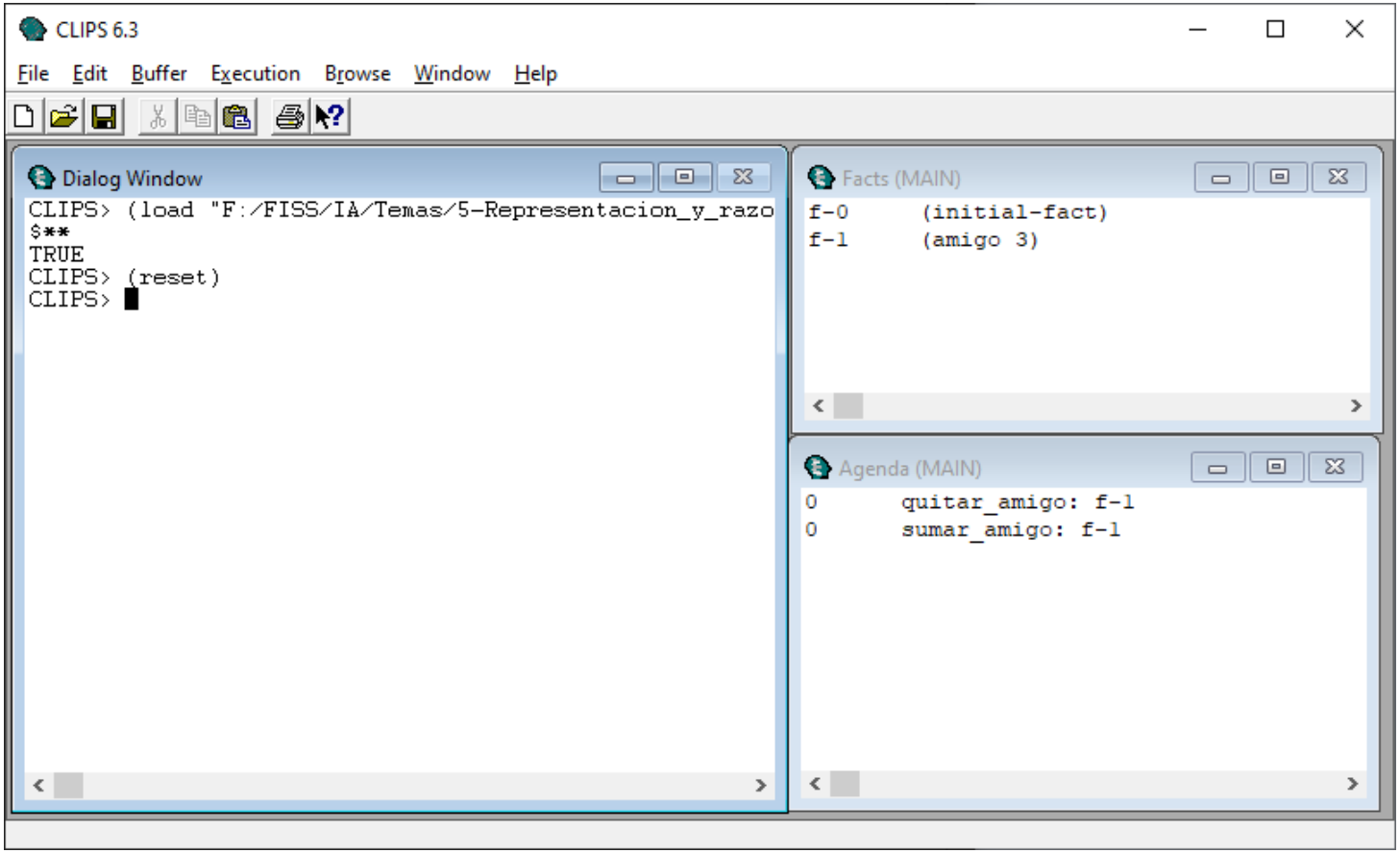
Motor de inferencia

- Qué regla se ejecutará primero?

```
(deffacts datos_iniciales
    (amigo 3)
)

(defrule quitar_amigo
  ?a <-(amigo ?x)
=>
    (retract ?a)
    (bind ?x (- ?x 1))
    (assert (amigo ?x))
)

(defrule sumar_amigo
  ?a <-(amigo ?x)
=>
    (retract ?a)
    (bind ?x (+ ?x 1))
    (assert (amigo ?x))
)
```



Reglas

- Ejemplo:
  - **Añadimos *(assert (color rojo))*** a la MT y creamos la siguiente regla:

```
(defrule cielo
   (color azul)
=>
   (assert (cielo-es despejado))
)
```

- Al lanzar la evaluación de las reglas en base a la lista de hechos, con el comando ***(run)***, si existiera el hecho ***(color azul)***, se añadiría un nuevo hecho a la lista ***(cielo-es despejado)***, circunstancia que podemos comprobar con ***(facts)*****,** que no sucede porque no existe.

```
CLIPS> (run)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (color rojo)
For a total of 2 facts.
```

Reglas

- Ejemplo:
  - **Añadimos** ahora el hecho ***(color azul)***, necesario para activar la regla, y ejecutamos

```
(CLIPS> (assert (color azul))
<Fact-2>
CLIPS> (run)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (color rojo)
f-2     (color azul)
f-3     (cielo-es despejado)
For a total of 4 facts.
```

- Tras ejecutar el programa en CLIPS, se añade un nuevo hecho a la MT: ***(cielo-es despejado)***
- Las reglas poseen refracción: **una regla sólo se dispara una vez para un conjunto dado de hechos**, con lo que se evitan bucles infinitos.

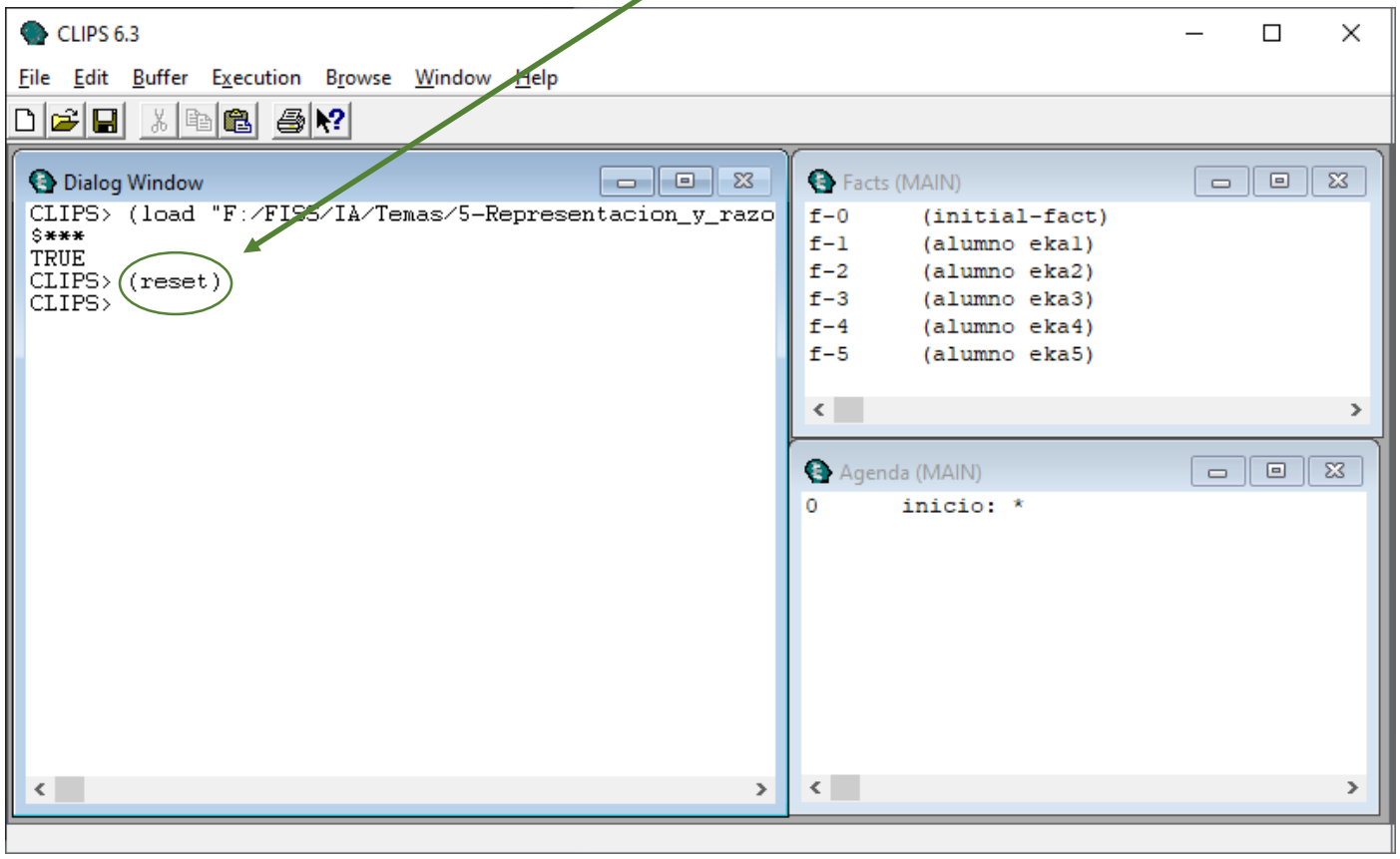
Ejemplo 1

- En la clase hay X alumnos y queremos saber cuantos hay:

```
(deffacts datos_iniciales
    (alumno eka1)
    (alumno eka2)
    (alumno eka3)
    (alumno eka4)
    (alumno eka5)
)
```

```
(defrule inicio
=>
    (assert (suma 0))
)
```

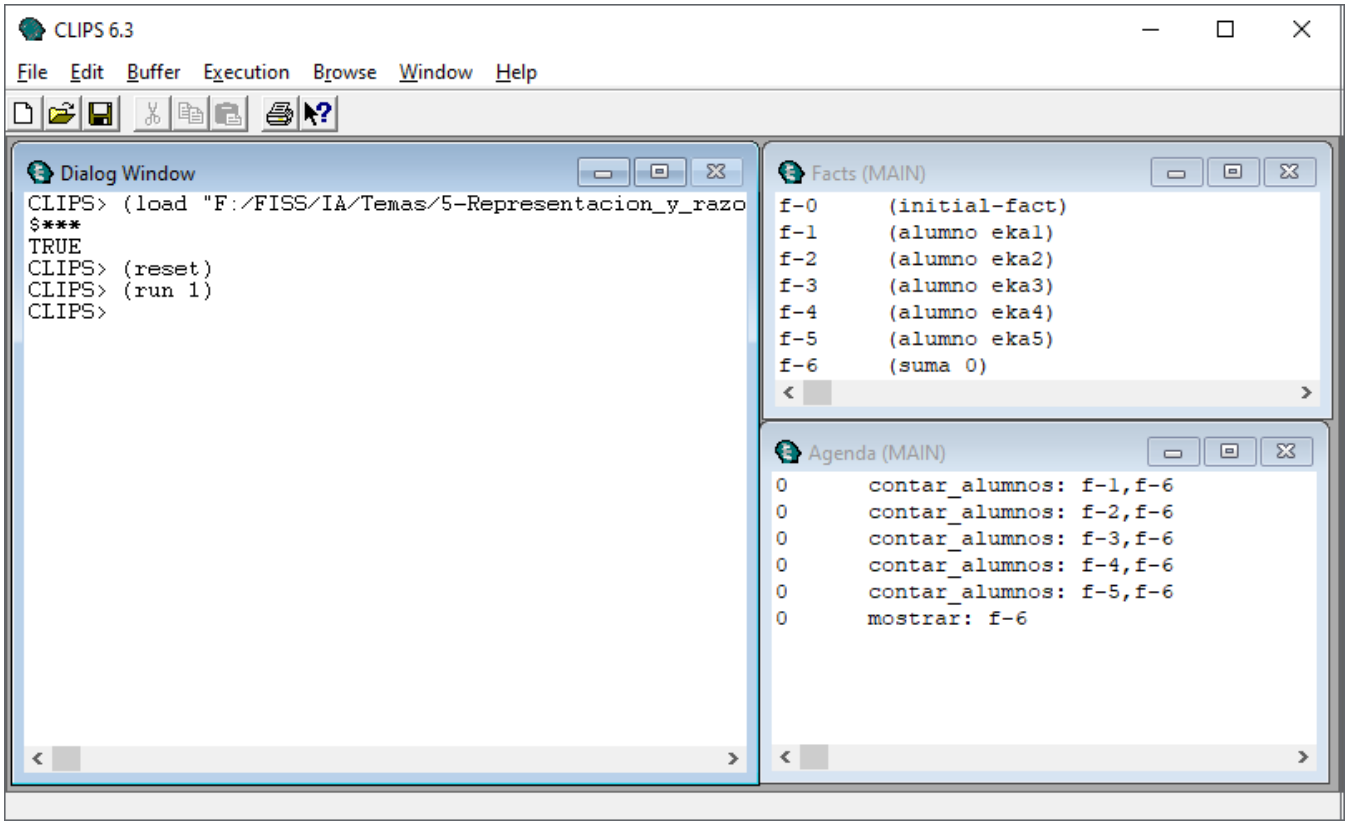
¡¡¡ IMPORTANTE !!!



Ejemplo 1

- En la clase hay X alumnos y queremos saber cuantos hay:

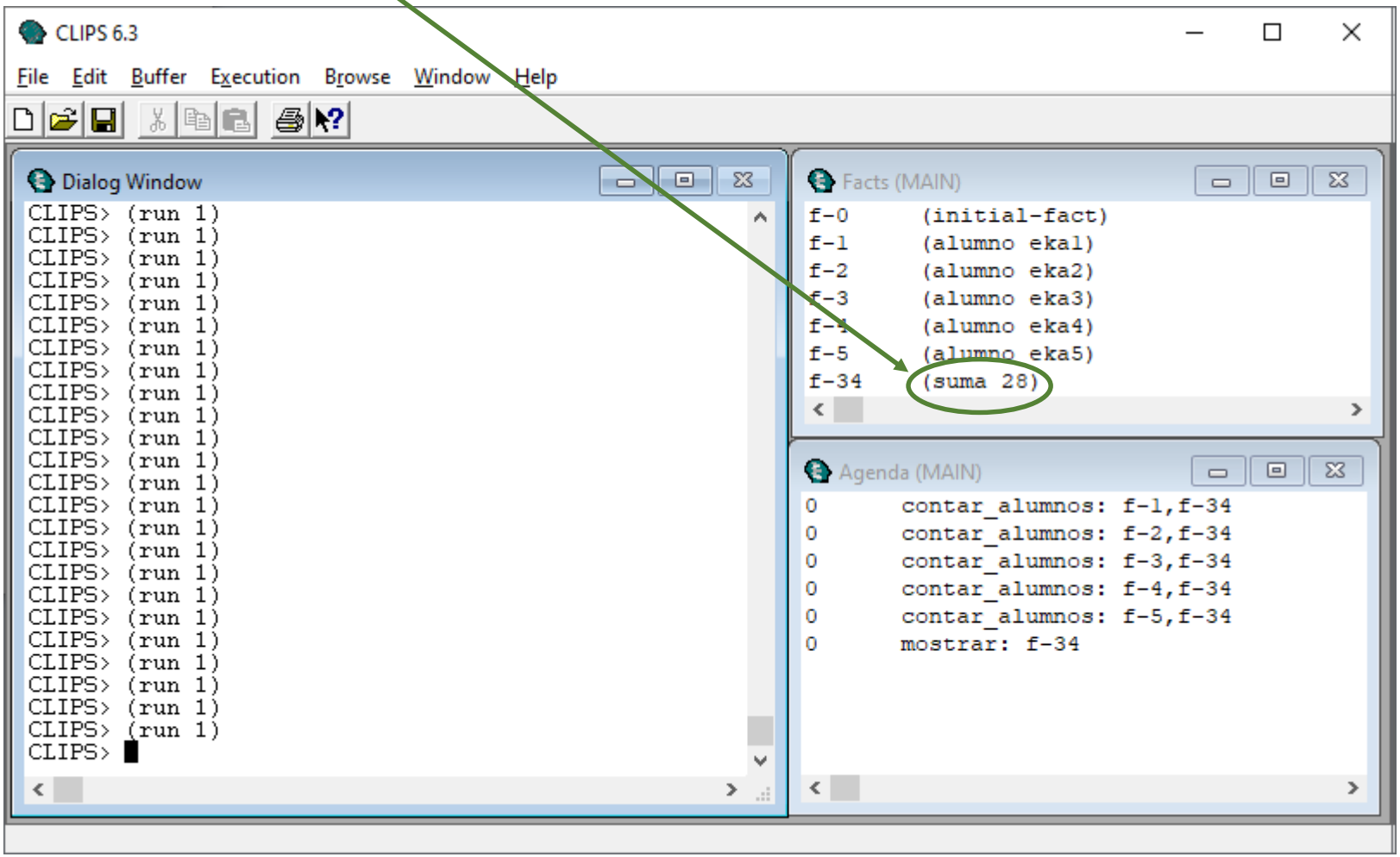
```
(defrule contar_alumnos
        (alumno ?)
?a <- (suma ?x)
=>
    (retract ?a)
    (bind ?x (+ ?x 1))
    (assert (suma ?x))
)
```




```
(defrule mostrar
    (suma ?x)
=>
    (printout t "Hay " ?x " alumnos" crlf)
)
```

Ejemplo 1

- Qué está ocurriendo?



Ejemplo 1

- En la clase hay X alumnos y queremos saber cuantos hay:

```
(deffacts datos_iniciales
    (alumno eka1)
    (alumno eka2)
    (alumno eka3)
    (alumno eka4)
    (alumno eka5)
)
```

```
(defrule contar_alumnos
?m <- (alumno ?)
?a <- (suma ?x)
=>
    (retract ?a ?m)
    (bind ?x (+ ?x 1))
    (assert (suma ?x))
)
```

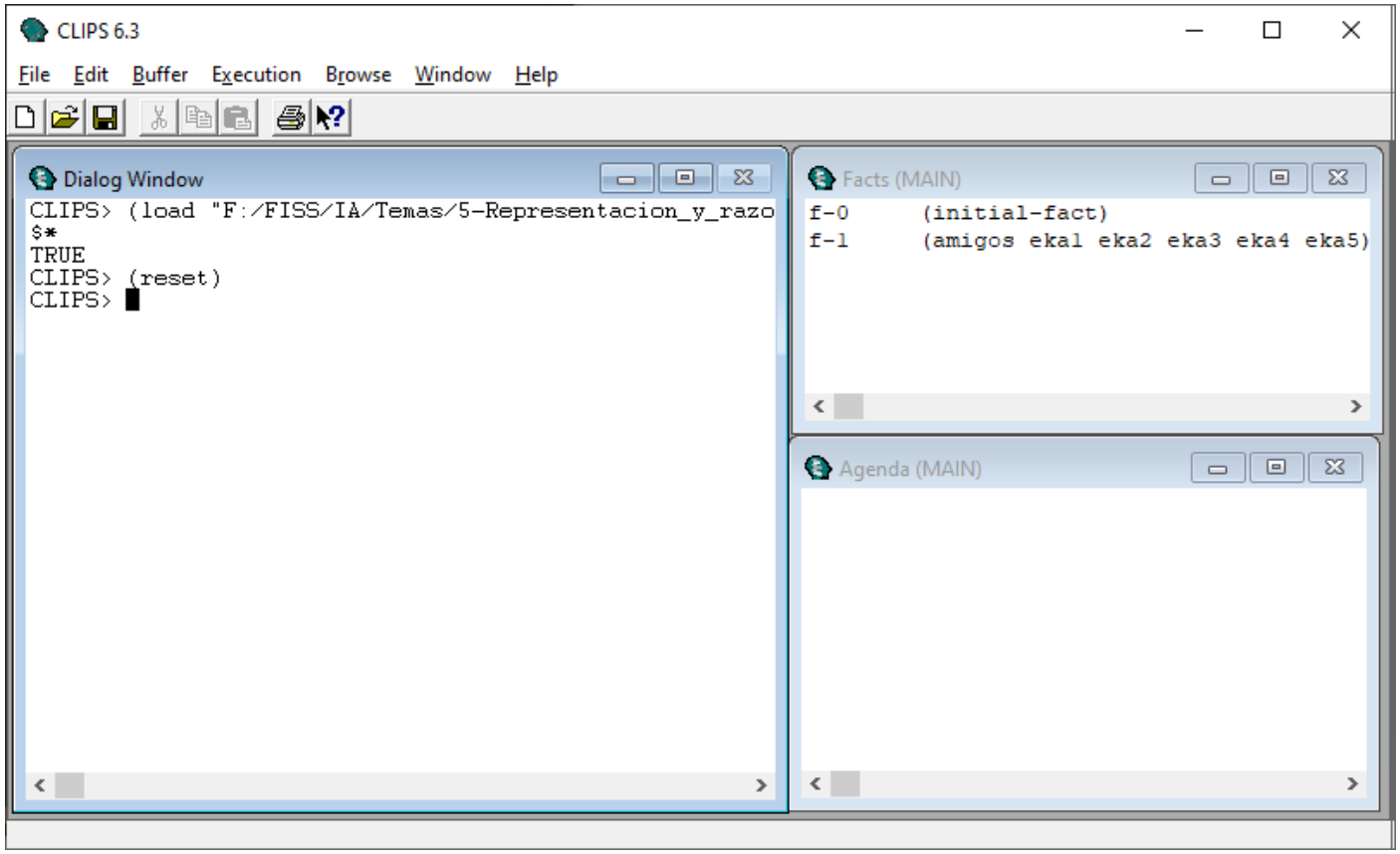
```
(defrule inicio
=>
    (assert (suma 0))
)
```

```
(defrule mostrar
    (suma ?x)
=>
    (printout t "Hay " ?x " alumnos" crlf)
)
```

Ejemplo 2

- Dada una lista de amigos y un nombre, indicar si es amigo o no

```
(deffacts datos_iniciales
    (amigos eka1 eka2 eka3 eka4 eka5 eka6 )
)
```

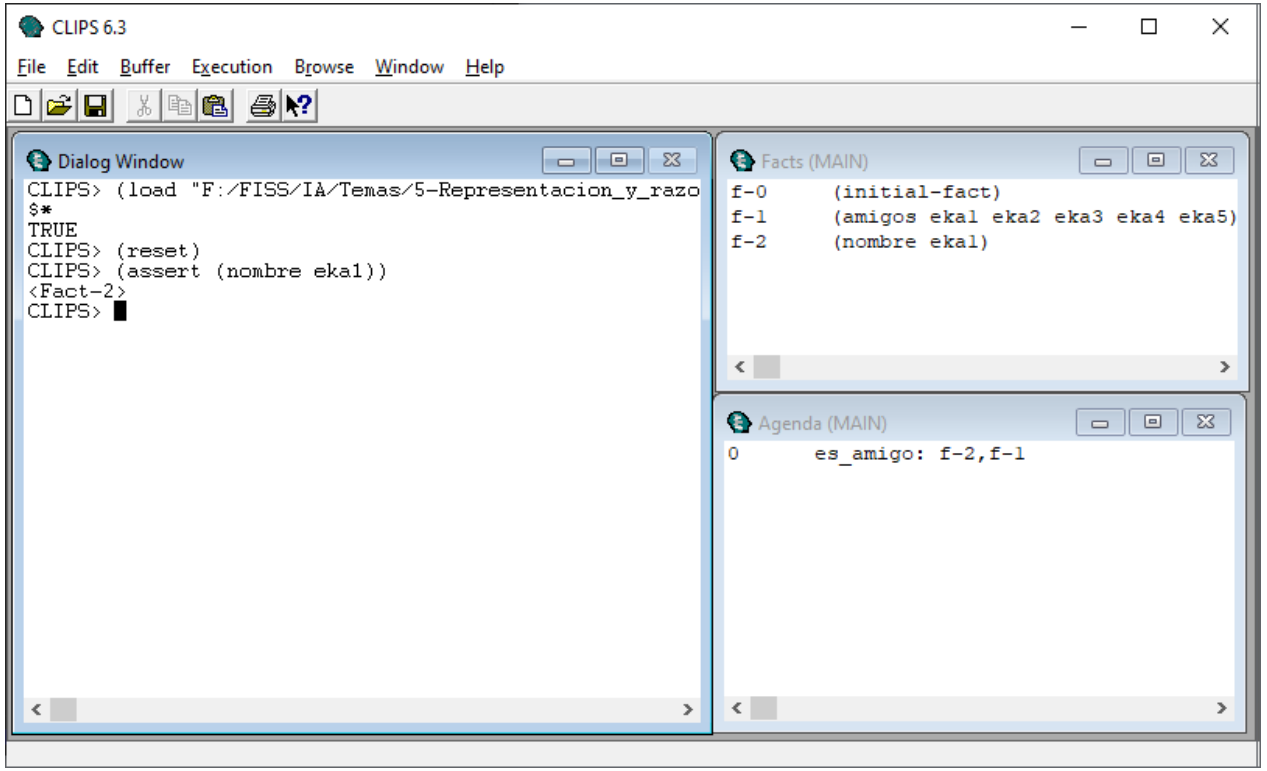


Ejemplo 2

- Dada una lista de amigos y un nombre, indicar si es amigo o no

```
(deffacts datos_iniciales
    (amigos eka1 eka2 eka3 eka4 eka5 eka6 )
)
```

```
(defrule es_amigo
    (nombre eka1)
    (amigos $? eka1 $?)
=>
    (printout t "es amigo mio" crlf)
)
```



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 1

```
(deffacts datos_iniciales
    (persona durmiendo)
)

(defrule estado_nuevo
    ?p<-(persona ?e)
=>
    (retract ?p)
    (printout t "cual es el nuevo estado?" crlf)
    (bind ?estado (read))
    (if (or(eq ?estado durmiendo)
           (eq ?estado estudiando)
           (eq ?estado comiendo)) then
        (assert (persona ?estado))
    else
        (printout t "ese estado no es posible" crlf)
        (assert (persona ?e))
    )
)
```

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 2

```
(deffacts datos_iniciales
    (fruta 5 5 5) ; platano-manzana-kiwi
    (dulce 5 5 5) ; tarta de queso-manzana-coulant
)

(defrule añadir_cliente
    (nuevo cliente ?nom)
=>
    (assert (cliente ?nom))
)


(defrule marchar_cliente
    (adios ?nom)
    ?p<-(cliente ?nom)
=>
    (retract ?p)
)
```

Ejercicio 2

```
(defrule platano ; lo mismo para los demás postres
    (cliente ?nom)
    ?h<-(quiere ?nom platano)
    ?f<-(fruta ?p1 ?p2 ?p3)
=>
    (retract ?h)
    (if (> ?p1 0) then
        (retract ?f)
        (assert (fruta (- ?p1 1) ?p2 ?p3))
    else
        (printout t "No queda platano")
    )
)

(defrule vacio
    ?f<-(dulce ?p1 ?p2 ?p3)
=>
    (if (and(= ?p1 0)(= ?p2 0)(= ?p3 0)) then
        (retract ?f)
    )
)
```

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.

Ejercicio 3

```
(deffacts datos_iniciales
    (pacman (random 1  4) (random 1  4))
    (fantasma (random 1  4) (random 1  4))
    (comida (random 1  4) (random 1  4))
    (turno pacman)
)

(defrule gana_pacman
    (pacman ?x ?y)
    (comida ?x ?y)
=>
    (printout t "gana pacman" crlf)
)

(defrule gana_fantasma
    (pacman ?x ?y)
    (fantasma ?x ?y)
=>
    (printout t "gana fantasma" crlf)
)
```

Ejercicio 3

```
(defrule mover_pacman_izquierda ; igual para el resto de movimientos
    ?p<-(pacman ?x ?y)
    ?m<-(mover izquierda)
    ?t<-(turno pacman)
=>
    (retract ?t ?m ?p)
    (assert(turno fantasma))
    (if (< 0 (- ?y 1)) then
        (assert(pacman ?x (- ?y 1)))
    )
)
```

Ejercicio 3

```
(defrule mover_fantasma ; 1 arriba, 2 abajo, 3 izquierda, 4 derecha
    ?p<-(fantasma ?x ?y)
    ?t<-(turno fantasma)
=>
    (retract ?t ?p)
    (assert(turno pacman))
    (bind ?n (random 1 4))
    (switch ?n
        (case 1 then
            (if (< (+ ?x 1) 4) then
                (assert (fantasma (+ ?x 1) ?y))
            ))
        (case 2 then
            (if (> (- ?x 1) 0) then
                (assert (fantasma (- ?x 1) ?y))
            ))
                    . . .
     )
)
```

Ejercicio 3

```
(case 3 then
    (if (< 0 (- ?y 1)) then
        (assert (fantasma ?x (- ?y 1)))
    ))
(case 4 then
    (if (< (+ ?y 1) 4) then
        (assert (fantasma ?x (+ ?y 1)))
    ))
```