3. Sistemas Basados en Conocimiento

3.1 Sistemas Expertos

Los **sistemas expertos** son un tipo específico de **SBC** cuyo objetivo es alcanzar un nivel de competencia igual o superior al de un experto humano en un dominio concreto (ejemplo: medicina).

Los **SBD** (Sistemas Basados en Conocimiento) es un término más general que engloba a los sistemas expertos y otros sistemas que usan conocimiento explícito.

Características:

- **Representación explícita del conocimiento:** el conocimiento relevante del dominio está formalizado y representado de forma explícita en el sistema
- **Razonamiento automático:** no basta con tener el conocimiento: el sistema debe tener un mecanismo sistemático para aplicar ese conocimiento (algoritmo de razonamiento), que depende de cómo se representa.

Algunos dominios son:

- **Diagnóstico médico**: apoyo a médicos para identificar enfermedades.
- **Análisis financiero**: ayuda a decidir inversiones según condiciones y riesgos.
- **Control y monitorización de procesos**: supervisión de sistemas industriales complejos.
- **Educación (tutorización inteligente)**: identificación de áreas de mejora en el aprendizaje.
- **Diagnóstico y mantenimiento de máquinas**: localización de fallos y planificación de mantenimiento.

Los **sistemas expertos** se deben se usar cuando trabajemos con **dominios complejos y especializados**, cuando los problemas requieren gran experiencia y conocimiento especializado. **Cuando no hay solución algorítmica clara**, si el problema no se puede resolver fácilmente con métodos clásicos. **Existe conocimiento experto disponible:** el conocimiento puede estar documentado o ser aportado por expertos humanos.

3.1.2 Diseñar un SE

Tenemos el rol de **ingeniero del conocimiento**, que es un profesional encargado de construir estos sistemas, obteniendo conocimiento útil sobre el problema y su

solución. Para ello:

- **Obtención del conocimiento:** extraer información relevante del dominio, generalmente de expertos humanos
- **Representación computacional:** formalizar ese conocimiento de forma que pueda ser procesado por una máquina (por ejemplo, reglas, hechos, ontologías).

El conocimiento evoluciona y mejora con el tiempo, por lo que los sistemas deben actualizarse y modificarse para seguir siendo útiles, lo que es el **problema principal de los SE**.

3.1.3 Como funciona un SE

Tiene los siguientes componentes:

- **Base de hechos:** información específica y concreta sobre el caso considerado (datos conocidos en un momento dado).
- **Base de conocimiento:** conjunto de reglas, información y experiencia útil para resolver problemas en el dominio específico
- **Mecanismo de razonamiento:** algoritmo que aplica el conocimiento (base de conocimiento) sobre los hechos para obtener una solución al problema

Funciona de la siguiente manera:

- El sistema compara los **hechos** (datos en particular) con el **conocimiento** (reglas, experiencia del dominio)
- El **mecanismo de razonamiento automático** permite llegar a una conclusión utilizando ambos bloques.

Por ejemplo:

- **Conocimiento:** si la temperatura corporal es superior a 37^oC, entonces el paciente tiene fiebre
- **Hecho:** la temperatura del paciente es de 38.3^oC
- **Conclusión (proceso de razonamiento):** el sistema aplica la reglas, verifica la condición y concluye: el paciente tiene fiebre

> **Nota**
>
> - Los **hechos** pueden cambiar con el tiempo (son dinámicos)
> - El **conocimiento** también puede evolucionar, por lo que el sistema debe ser adaptable y actualizable

3.1.4 Estructura de un SE

Base de conocimiento, contiene las reglas del tipo **SI** (antecedentes), **ENTONCES** (consecuente). Ejemplo:

- SI temperatura > 37°C, ENTONCES el paciente tiene fiebre.

Base de hechos, almacena los datos y hechos conocidos sobre el caso. Incluye:

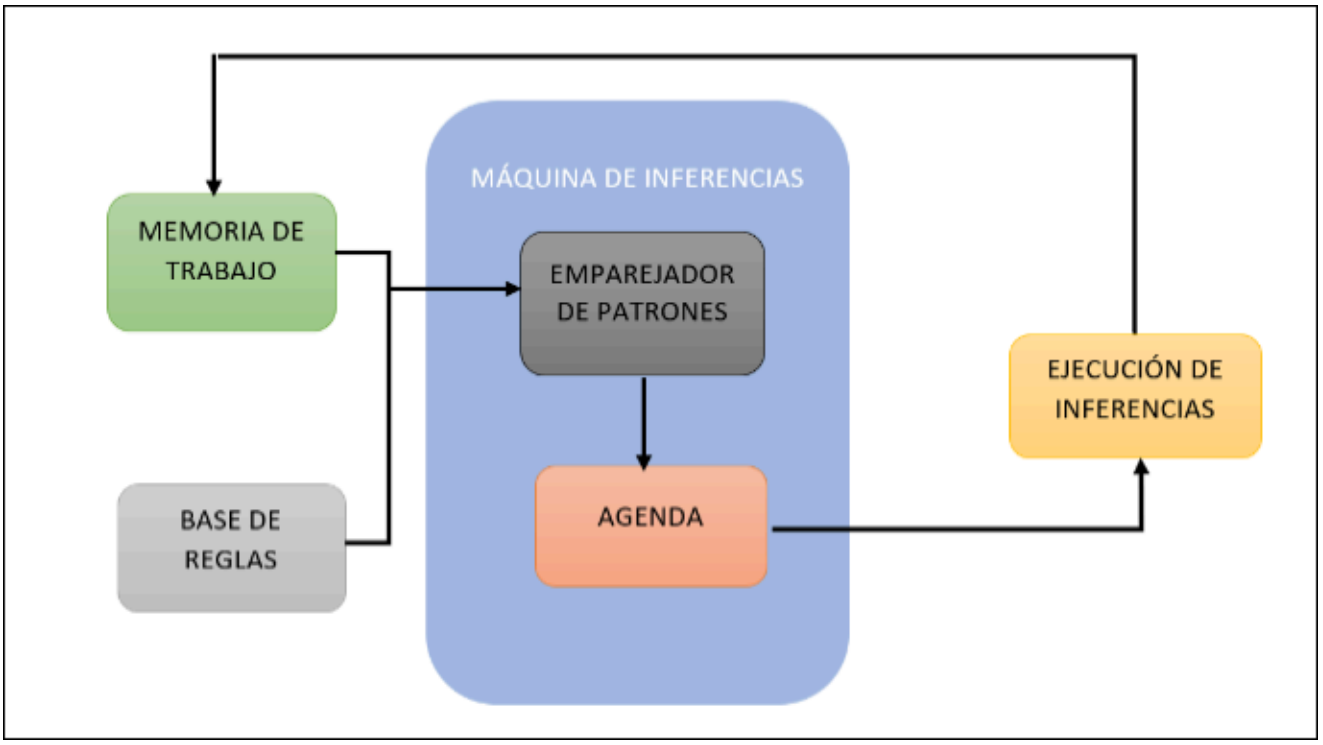
- Datos y hechos actuales.
- Metas a alcanzar (objetivos).
- Hipótesis avanzadas (suposiciones o posibles explicaciones).
- Lista de subproblemas.
- Reglas pendientes de aplicar.

Motor de inferencias, es el “cerebro” del sistema experto; se encarga de aplicar las reglas al caso concreto. Funciones principales:

- **Equiparación**: buscar qué reglas pueden aplicarse según los hechos conocidos.
- **Resolución de conflictos**: decidir qué reglas aplicar cuando hay varias posibles.

Interfaz de usuario, permite la comunicación entre el usuario (persona que consulta el sistema) y el sistema experto.

Fuentes externas de datos, permiten que el sistema reciba información adicional de otras fuentes (sensores, bases de datos, etc).



Memoria de trabajo. equivalente a la base de hechos; almacena los datos actuales del problema.

Base de reglas, equivalente a la base de conocimiento; contiene todas las reglas del sistema.

Emparejador de patrones, componente que compara los hechos (memoria de trabajo) con las reglas para ver cuáles pueden aplicarse.

Agenda, lista de reglas que pueden aplicarse en el momento actual (candidatas).

Ejecutador de inferencias, aplica la regla seleccionada de la agenda, cambiando o añadiendo hechos; así el sistema avanza en la resolución del problema.

Resumen:

1. El usuario introduce datos a través de la interfaz.
2. El sistema recoge los hechos y consulta la base de conocimiento.
3. El motor de inferencias compara los hechos con las reglas disponibles.
4. Si se pueden aplicar varias reglas, resuelve el conflicto y escoge una.
5. Se ejecuta la inferencia, se actualizan los hechos, y el proceso se repite hasta obtener una conclusión o solución.

3.2 Sistemas Basados en Reglas

Las **reglas de producción** son una forma típica de representar conocimiento entre sistemas cuya estructura es **SE** (situación/condición) **ENTONCES** (acción/conclusión)

Una **situación** es un conjunto de condiciones que se deben cumplirse. **Acciones típicas:**

- Añadir hechos a la memoria de trabajo (espacio donde se almacenan hechos conocidos e inferidos)
- Suprimir hechos de la memoria de trabajo
- Ejecutar procedimientos
- Imprimir elementos o solicitar datos

3.2.1 Funcionamiento

- Parten de los hechos iniciales en la memoria de trabajo.
- Se emparejan hechos con antecedentes de las reglas.
- Se aplican reglas hasta alcanzar un objetivo o no poder continuar.
- Se seleccionan reglas aplicables dentro del “conjunto conflicto” (reglas que pueden aplicarse en ese momento).

Principio de refracción: No se aplica reiteradamente una misma regla si ya se ha aplicado.

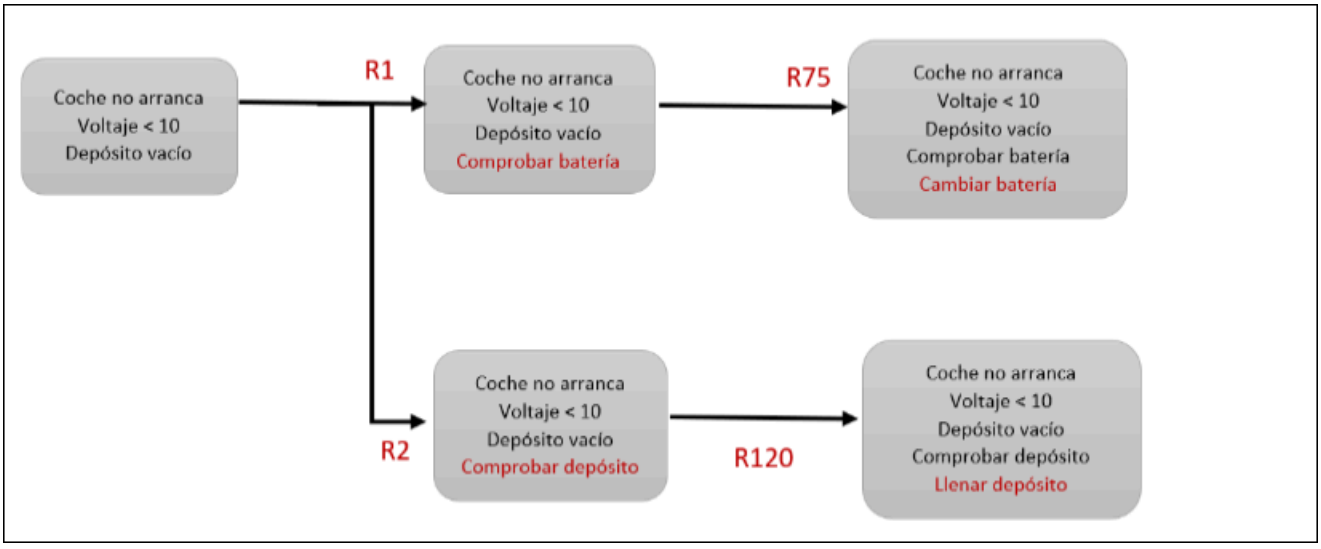
3.2.2 Encadenamiento de reglas

Encadenamiento hacia adelante (Forward Chaining)

- Las reglas son independientes y se pueden aplicar en cualquier orden, siempre que se verifiquen las condiciones.
- Aunque independientes, pueden estar relacionadas si la conclusión de una es condición de otra.
- Se aplican sucesivamente reglas que coincidan con los hechos presentes, avanzando hasta alcanzar una meta.

Ejemplo:

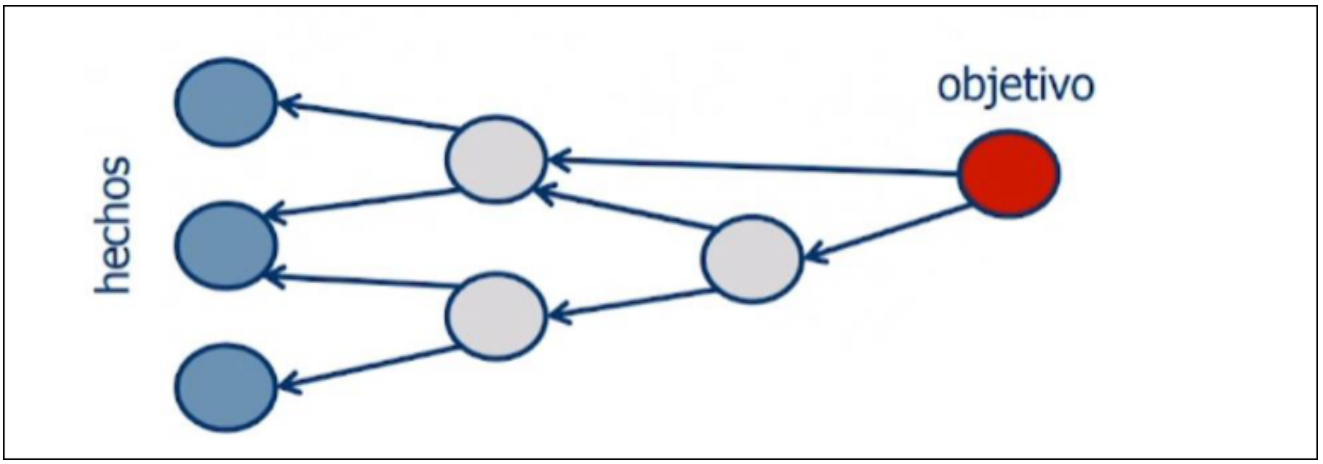
```
Regla 1. IF coche no arranca,
         THEN comprobar batería
Regla 2. IF coche no arranca
         THEN comprobar combustible
...
Regla 75. IF comprobar batería
          AND voltaje batería < 10V
          THEN cambiar batería
...
Regla 120. IF comprobar combustible
           AND depósito de combustible vacío
           THEN llenar depósito.
```

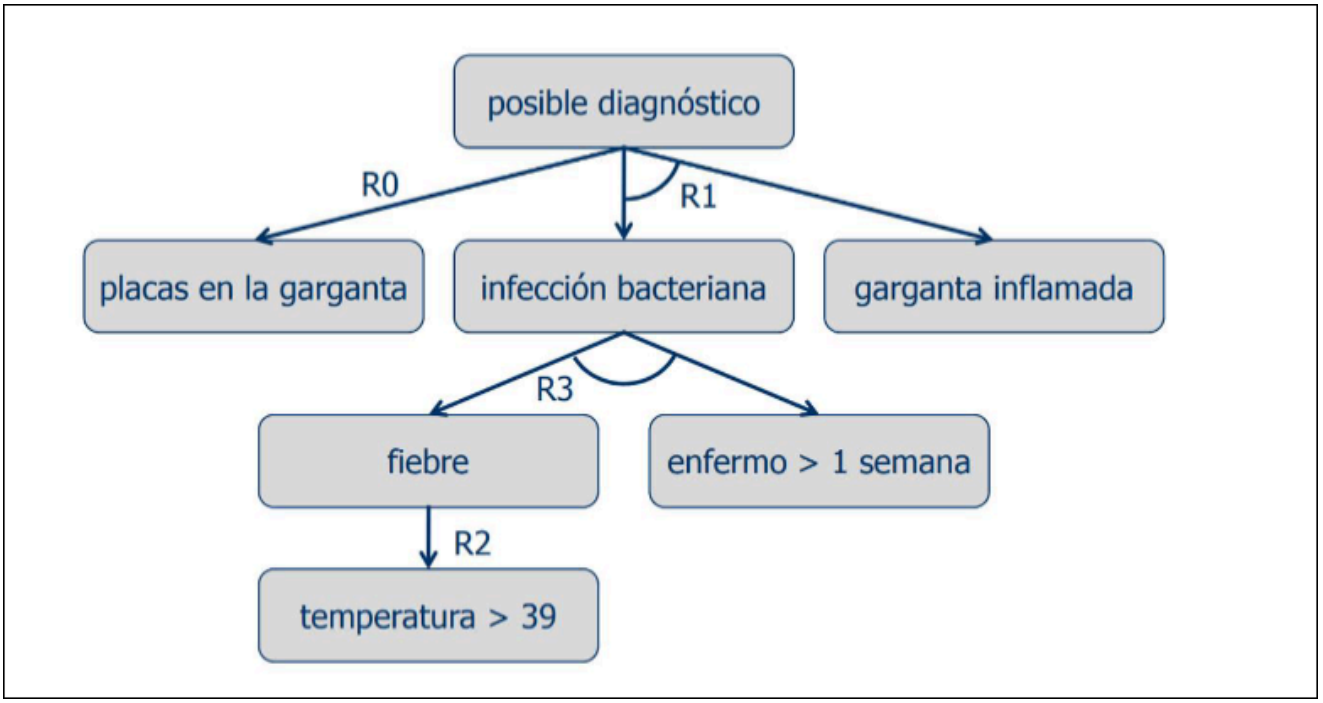


Encadenamiento hacia atrás (Backward Chaining)

- Usado habitualmente en sistemas expertos cuando se busca responder una pregunta concreta.
- El razonamiento parte de la meta (lo que se quiere saber) y busca verificar si las condiciones necesarias se cumplen, explorando hacia atrás en las reglas.
- Utiliza estructuras tipo grafo para representar relaciones entre reglas (AND, OR, NOT).

Ejemplo: Si queremos saber si un paciente tiene infección de garganta, se verifica si cumple las condiciones de varias reglas relacionadas.





3.2.3 Pseudocódigos

Método básico de razonamiento:

```
1: BH = HechosIniciales;
2: mientras NoVerificaCondiciónFinalización(BH) o NoseEjecutaAcciónde Parada hacer
3:    ConjuntoConflicto = Equiparar(BC,BH);
4:    R=Resolver(ConjuntoConflicto);
5:    NuevosHechos = Aplicar(R,BH);
6:    Actualizar(BH,NuevosHechos);
7: fin mientras
```

Verificación de reglas:

```
1: Verificado=Falso;
2: si Contenida (Meta,BH) entonces
3:    devolver Verdadero;
4: si no
5:    ConjuntoConflicto = Equiparar(Consecuentes(BC),Meta);
6:    mientras NoVacío(ConjuntoConflicto) y No(Verificado) hacer
7:       R=Resolver(ConjuntoConflicto);
8:       Eliminar(R,ConjuntoConflicto);
9:       NuevasMetas=ExtraerAntecedentes(R), Verificado=Verdadero;
10:      mientras NoVacío(NuevasMetas) y Verificado hacer
11:         Meta=SeleccionarMeta(NuevasMetas);
12:         Eliminar(Meta, NuevasMetas);
13:         Verificado=Verificar(Meta,BH);
14:         si Verificado entonces
15:            Añadir(Meta,BH);
16:         fin si
17:      fin mientras
18:   fin mientras
19:   devolver(Verificado);
20: fin si
```

Encadenamiento hacia adelante:

```
1: BH = HechosIniciales, ConjuntoConflicto = ExtraeCualquierRegla(BC);
2: mientras NoContenida(Meta,BH) y NoVacío(ConjuntoConflicto) hacer
3:    ConjuntoConflicto = Equiparar(Antecedentes(BC),BH);
4:    si NoVacío(ConjuntoConflicto) entonces
5:       R=Resolver(ConjuntoConflicto);
6:       NuevosHechos = Aplicar(R,BH);
7:       Actualizar(BH,NuevosHechos);
8:    fin si
9: fin mientras
10: si Contenida(Meta,BH) entonces
11:    devolver "éxito";
12: fin si
```

Encadenamiento hacia atrás:

```
1: BH = HechosIniciales;
2: si Verificar (Meta,BH) entonces
3:     devolver "éxito";
4: si no
5:     devolver "fracaso";
6: fin si
```

3.3 Conocimientos vs Hechos

Hechos	Conocimiento
Específicos, relacionados con el problema concreto que se intenta resolver.	General, relacionado con el dominio en el que se opera y el tipo de problemas.
Dinámicos, a partir de unos hechos iniciales pueden aparecer otros distintos.	Relativamente estático, aunque puede cambiar si se añade nuevo conocimiento.
Aumentan durante la resolución del problema.	Normalmente no aumenta durante la resolución del problema.
Necesidad de almacenamiento y recuperación eficiente.	Necesidad de razonamiento eficiente.
Se busca que los hechos obtenidos directamente del problema sean precisos y ciertos.	Puede ser impreciso e incierto; por lo tanto, también los hechos inferidos pueden serlo.

3.4 Programación convencional vs Sistemas expertos

Programación convencional	Sistemas expertos
Programación imperativa, instrucciones en orden, cada línea de código es una acción o mandato explícito.	Programación declarativa, conjunto de reglas potencialmente útil, pero no es necesario evaluar todas ni en un orden determinado.
Modificación mediante reprogramación, hay que programar y recompilar.	Modificación en la base de conocimiento; existe una base de datos única y al cambiar las reglas no es necesario recompilar.
Solución algorítmica.	Solución por razonamiento basado en conocimiento. Se aplican a problemas complejos.
Normalmente la solución es precisa y óptima.	Normalmente la solución es imprecisa, con márgenes de certidumbre (solución probable o posible).
La ejecución está guiada por el flujo de ejecución del código.	La ejecución está guiada por el motor de inferencia o el mecanismo de razonamiento automático.

3.5 CLIPS

CLIPS (C Language Integrated Production System) es una herramienta/software para desarrollar **sistemas expertos** y **sistemas basados en reglas**.

3.5.1 Conceptos básicos

CLIPS maneja tres conceptos clave:

- **Hechos:** Son datos sobre el estado del mundo o del sistema. Ejemplo:

```
(assert (temperatura 40))
```

- **Reglas:** Son inferencias del tipo **SI** (condiciones) **ENTONCES** (acciones). Ejemplo:

```
(defrule fiebre
   (temperatura ?t&:(> ?t 39))
   =>
   (assert (fiebre)))
```

Esta regla dice: si la temperatura es mayor que 39, entonces se considera que hay fiebre.

- **Memoria de trabajo:** es donde CLIPS guarda todos los hechos conocidos y los va modificando según se aplican reglas.

3.5.2 Cómo usar CLIPS

- **Crear hechos:** en CLIPS, podemos introducir hechos directamente:

```
(assert (garganta_inflamada))
(assert (temperatura 40))
```

- **Definir reglas:** se usan bloques `defrule`:

```
(defrule diagnostico_fiebre
   (temperatura ?t&:(> ?t 39))
   =>
   (assert (fiebre)))
```

Otra regla que depende de la anterior:

```
(defrule diagnostico_infeccion
   (garganta_inflamada)
   (fiebre)
   =>
   (assert (infeccion_bacteriana)))
```

- **Ejecutar el motor de inferencia:** después de definir hechos y reglas, ejecuta:

```
(reset)   ; Limpia y prepara la memoria de trabajo
(run)     ; Ejecuta las reglas hasta que no se pueda inferir nada más
```

- **Ver los resultados:** puedes consultar los hechos inferidos:

```
(facts)
```

Esto mostrará todos los hechos en la memoria de trabajo, incluidos los inferidos por las reglas.

3.5.3 Ejemplo completo sencillo

Supongamos que queremos diagnosticar si una persona tiene una infección por fiebre y garganta inflamada:

```
; Hechos iniciales
(assert (temperatura 40))
(assert (garganta_inflamada))

; Reglas
(defrule fiebre
   (temperatura ?t&:(> ?t 39))
   =>
   (assert (fiebre)))

(defrule infeccion
   (garganta_inflamada)
   (fiebre)
   =>
   (assert (infeccion_bacteriana)))

; Ejecutar
(reset)
(run)
(facts)
```

¿Qué ocurre?

1. CLIPS ve que hay temperatura de 40, aplica la regla de fiebre.
2. Al tener fiebre y garganta inflamada, aplica la regla de infección bacteriana.
3. Si ejecutas `(facts)` verás que tienes los hechos “fiebre” e “infeccion_bacteriana”.

3.5.4 Sintaxis Hechos

- **Vectores ordenados de características:** forma simple de representar información como una lista ordenada de valores, cada uno se asocia a un atributo por su posición.

  ```
  (assert (Pedro 45 V NO))
  ```

  CLIPS asigna automáticamente un ID (por ejemplo, f-0, f-1...) cuando se agrega un hecho.
- **Registros (Templates):** son estructuras de datos con campos nombrados, mucho más claros y flexibles que los vectores. Se usa `deftemplate` para definir la estructura:

```
(deftemplate Persona
   (field Nombre)
   (field Edad)
   (field Sexo)
   (field EstadoCivil) )

(assert (Persona (Nombre Juan) (Edad 30) (EstadoCivil casado) (Sexo V)))
```

Cada valor se asigna explícitamente por nombre, no por orden.
Si no se da un valor para un campo, CLIPS pone `nil` (valor nulo).
Si necesitas que un campo contenga más de un valor (como varios nombres), usas `multifield`:

```
(deftemplate Persona (multifield Nombre) (field Edad) (field Sexo)
(field EstadoCivil))
(assert (Persona (Nombre Juan Carlos Cubero) (Edad 30)))
```

- **Carga de hechos desde archivo**

```
(load personas.clp)
```

- Para que se añadan a la memoria de trabajo, ejecuta:

```
(reset)
```

  - Esto borra los hechos anteriores, añade los nuevos hechos del fichero, pero mantiene los templates (registros).

- **Eliminación de hechos**

```
(retract 1 2) ; elimina los hechos f-1 y f-2
(retract *); borra todos los hechos
```

- **Modificación de hechos**

```
(modify 1 (Edad 27)) ; cambia la edad del hecho f-1 a 27
```

  - El identificador puede cambiar tras modificar (por ejemplo, de f-1 a f-4).

- **Duplicación de hechos**

```
(duplicate 1 (Nombre "Pedro Pérez") (Edad 40))
```

- **Eliminación completa**

```
(clear); Borrar todos los hechos, patrones y reglas
```

Ejemplo práctico resumido

```
(deftemplate Persona
   (field Nombre)
   (field Edad)
   (field Sexo)
   (field EstadoCivil))

(assert (Persona (Nombre Ana) (Edad 25) (Sexo M) (EstadoCivil soltera)))
(facts) ; muestra hechos actuales

(modify 1 (Edad 26)) ; cambia la edad del hecho f-1
(duplicate 1 (Nombre "Ana López")) ; copia el hecho cambiando el nombre
(retract *) ; borra todos los hechos
(clear) ; borra todo, incluyendo reglas y templates
```

3.5.5 Sintaxis Reglas

Una **regla** en CLIPS es una instrucción que indica:
"Si se cumplen ciertas condiciones sobre los hechos en la memoria de trabajo, entonces realiza una o varias acciones."

```
(defrule <nombre> <comentario opcional>
  <patrones en el antecedente>
  =>
  <acciones en el consecuente>
)
```

- **Antecedente** (LHS, Left Hand Side):
  Las condiciones o patrones que deben cumplirse para activar la regla.
- **Consecuente** (RHS, Right Hand Side):
  Las acciones que se ejecutan si se cumple el antecedente, como añadir hechos (`assert`), eliminarlos (`retract`), imprimir información (`printout`), etc.

Supón que tienes estos hechos:

```
(assert (temperatura 40))
(assert (garganta_inflamada))
```

Y defines la siguiente regla:

```
(defrule diagnostico-fiebre
   ; Si la temperatura es mayor a 39
   (temperatura ?t&:(> ?t 39))
   =>
   ; Entonces, añade el hecho "fiebre"
   (assert (fiebre))
)
```

- **(printout t "texto")**: Imprime mensajes por pantalla.
- `(run 1)`: Ejecuta una regla (o el número que indiques).
  Puedes repetir `(run)` para seguir ejecutando reglas.

3.5.6 Patrones y Variables

- **Patrones:** Son las condiciones que aparecen en la parte izquierda (LHS) de una regla, y sirven para buscar hechos en la memoria de trabajo que "encajen" con esos patrones.
- **Variables:** Permiten capturar valores de los hechos para usarlos en la parte derecha (RHS) de la regla o para imponer restricciones adicionales.
- El comodín `$?` indica que puede haber "cero o más" elementos en esa posición.
- Permite que una regla se active varias veces con el mismo hecho, dependiendo de cómo se emparejen los patrones.

Ejemplo:

```
(defrule ejemplo-emergencia
  (Emergencia ?sector)
  =>
  (printout t "Emergencia -> " ?sector crlf))
```

Si tienes varios hechos `(Emergencia A)`, `(Emergencia B)`, `(Emergencia C)`, la regla se activará una vez por cada hecho.

- Cuando una regla tiene varios patrones en la parte LHS, CLIPS exige que **todos** se cumplan para que se active la regla.
- Es decir, hay un "AND" implícito entre los patrones.
- Puedes imponer restricciones lógicas (condiciones) sobre los valores de los campos usando operadores como `&` (AND), `|` (OR), y otros operadores lógicos o de comparación.
- Ejemplo: `(Edad ?e&:(> ?e 18))` solo empareja hechos con edad mayor a 18.

Ejemplo práctico de registros y patrones

```
;Definición de plantilla:
(deftemplate FichaPaciente
  (field Nombre)
  (field Casado)
  (field Direccion))

;Inserción de hecho:
(assert (FichaPaciente (Nombre Juan) (Casado No) (Direccion "Santiago de
Compostela")))

;Regla usando patrón y variables:
(defrule mostrar-direccion
  (FichaPaciente (Nombre ?nombre) (Direccion ?direccion))
  =>
  (printout t ?nombre " vive en " ?direccion crlf))
```