Modularización

En los sistemas expertos de diagnóstico se distinguen distintas fases:

1. **Detección del problema**: ¿Qué ha fallado?
2. **Diagnóstico**: ¿Por qué ha fallado?
3. **Resolución** (terapia): ¿Cómo resolver ó solucionar el fallo?

Para conseguir que primero se disparen las reglas de detección, luego las de diagnóstico y, finalmente las de terapia, podemos utilizar distintas estrategias:

- Asignarles a todas las reglas de una fase la misma prioridad, pero mayor que la prioridad de las reglas de una fase posterior.
- Utilizar hechos de control.
- Definir módulos de CLIPS.

Modularización con hechos de control

EJEMPLO: Diagnóstico médico.

```
(defrule LeerSintomas
      ?fase <- (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca los síntomas de " ?Nombre " separados
      por espacios y ordenados alfabéticamente" crlf)
      (retract ?fase))

(defrule DiagnosticoEccema
      (Sintomas ?Nombre $? picor $? vesiculas $?)
      =>
      (printout t "El diagnóstico para " ?Nombre " es: Eccema" crlf))


CLIPS> (assert (fase LecturaSintomas Pedro))
CLIPS> (run)
  Introduzca los síntomas de Pedro separados por espacios y
  ordenados alfabéticamente
CLIPS> (assert (Sintomas Pedro picor vesiculas))
CLIPS> (run)
  El diagnóstico para Pedro es: Eccema
```

Podemos evitar que el usuario tenga que introducir órdenes específicas de CLIPS:

```
; Fase inicial

(defrule ObtenerNombreUsuario
      =>
      (printout t "Introduzca su nombre" crlf)
      (bind ?Respuesta (read))
      (assert (fase LecturaSintomas ?Respuesta)))


; Fase de lectura de síntomas

(defrule LeerSintomas
      ?fase <- (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca los síntomas de " ?Nombre " separados
      por espacios y ordenados alfabéticamente" crlf)
      (bind ?Respuesta (explode$ (readline)))
      (assert (Sintomas ?Nombre ?Respuesta))
      (retract ?fase)
)


; Fase de diagnóstico

(defrule DiagnosticoEccema
      (Sintomas ?Nombre $? picor $? vesiculas $?)
      =>
      (printout t "El diagnóstico para " ?Nombre " es: Eccema" crlf))
```

```
CLIPS> (reset)
CLIPS> (run)
  Introduzca su nombre
  Pedro

  Introduzca los síntomas de Pedro separados por espacios y
  ordenados alfabéticamente

  picor vesículas

  El diagnóstico para Pedro es: Eccema
```

El problema se complica cuando tenemos que manejar más fases y hay varias reglas en una misma fase. Por ejemplo, puede que tengamos reglas que nos dicen si una persona es o no adulta, si tiene o no fiebre… y deseemos que estas reglas se lancen antes que cualquier otra regla de diagnóstico o de terapia.

```
; Fase inicial

(defrule ObtenerNombreUsuario
      =>
      (printout t "Introduzca su nombre" crlf)
      (bind ?Respuesta (read))
      (assert (fase LecturaSintomas ?Respuesta)))


; Fase de lectura de síntomas

(defrule ObtenerTemperaturaUsuario
      (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca temperatura" crlf)
      (bind ?Respuesta (read)))

(defrule ObtenerSintomasUsuario
      (fase LecturaSintomas ?Nombre)
      =>
      (printout t "Introduzca los síntomas de " ?Nombre " separados
      por espacios y ordenados alfabéticamente" crlf)
      (bind ?Respuesta (explode$ (readline)))
      (assert (Sintomas ?Nombre ?Respuesta)))

…

; Última regla en aplicarse

(defrule CambiarDeFase
      (declare (salience -10))
      ?fase <- (fase LecturaSintomas ?Nombre)
      =>
      (retract ?fase)
      (assert (fase Diagnostico ?Nombre)))
```

Lo mismo se haría con los conjuntos de reglas de diagnóstico, terapia, salida de resultados, etcétera.

Por ejemplo, en todas las reglas relativas al diagnóstico, tendré como patrón en el antecedente `(fase Diagnostico)`

```
(defrule PacienteConFiebre
      (fase Diagnostico ?N)
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (> ?T 37))
      =>
      (assert (Paciente ?N TieneFiebre)))

(defrule PacienteSinFiebre
      (fase Diagnostico ?N)
      (ExploracionPaciente
            (Nombre           ?N)
            (Temperatura      ?T))
      (test (<= ?T 37))
      ?PF <- (Paciente ?N TieneFiebre)
      =>
      (retract ?PF))

(defrule DiagnosticoEccema
      (fase Diagnostico ?N)
      (Sintomas ?N $? picor $? vesiculas $?)
      =>
      (assert (PosibleDiagnostico ?N Eccema)))
…
```

En vez de usar una regla para cambiar de fase en cada una de ellas, podemos usar una única regla, que además nos permitirá volver a empezar con la primera fase una vez concluida la última:

```
(deffacts InformacionControl
       (fase Deteccion)
       (SecuenciaFases Diagnostico Terapia Deteccion))

(defrule CambiarDeFase
      (declare (Salience -10))
      ?fase <- (fase ?)
      ?ListaFases <- (SecuenciaDeFases  ?Siguiente $?Resto)
      =>
      (retract ?fase ?ListaFases)
      (assert (fase ?Siguiente))
      (assert (SecuenciaDeFases $?Resto ?Siguiente)))
```

Supongamos que estamos en una fase cualquiera, por ejemplo `Diagnóstico`. Cuando ya no existan reglas aplicables con `(fase Diagnostico)` en su antecedente, se activará la regla `CambiarDeFase` que ya estaba en la agenda a la espera de ejecutarse (la última debido a su baja prioridad).

El único problema que se puede presentar es que si ejecutamos `(run)`, podríamos entrar en un ciclo infinito, ya que procedería a cambiar ininterrumpidamente de fase.

Sobre el uso de prioridades en un sistema experto

Normalmente, no resulta recomendable utilizar más de 4 ó 5 prioridades diferentes (aunque CLIPS admita cualquier entero entre `-10.000` y `+10.000`), ya que un uso excesivo de prioridades conduce a una programación imperativa y no declarativa.

Normalmente, de mayor a menor prioridad, esta es la forma en que se agrupan las reglas de un sistema experto:

- **Restricciones**: Entrada de valores no permitidos y detección de estados que violan alguna restricción de nuestro problema.

- **Experto**: Las reglas usuales del sistema experto, que son de mayor prioridad que las reglas relacionadas con la realización de preguntas al usuario (para no ir preguntándole a éste si la respuesta puede obtenerse con las reglas del sistema).

- **Preguntas**: Lectura de datos proporcionados por el usuario.

- **Control**: Por último, las reglas con menor prioridad corresponden a las reglas que gobiernan las transiciones entre distintas fases.

Modularización con (defmodule)

`(defmodule)` se usa en CLIPS para dividir la base de conocimiento en distintos módulos. Cada módulo tendrá una agenda y una memoria de trabajo diferente, por lo que nos servirán para controlar las distintas fases de un sistema experto.

```
(defmodule  <Nombre del Módulo>
      (export  <Construcciones que exporta>)
      (import  <Construcciones que importa>)
      …
)
```

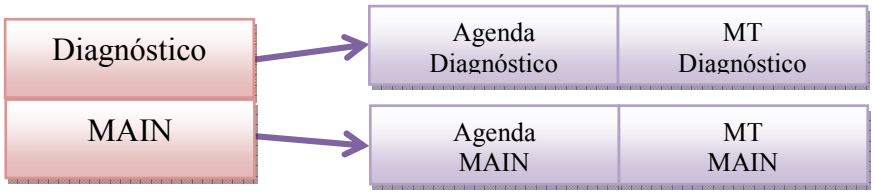
Todo lo que escribamos a partir de `(defmodule MODULO …)` pertenece a dicho módulo y, en principio, no es accesible desde otros módulos.

```
(defmodule Deteccion)
      (defrule TomarDatos …)
      …
)

(defmodule Diagnostico
      (defrule Eccema …)
      …
)

(defmodule Terapia
      …
)
```

CLIPS utiliza una pila de módulos para gestionar el módulo que está activo en cada momento, a la que denomina "*focus stack*" (de forma similar a como un lenguaje de programación imperativo útiliza una pila para gestionar las llamadas a subprogramas):



Cuando usamos `(run)`, se ejecutan las reglas del último módulo de la lista (`Diagnóstico` en la figura) y, cuando su agenda se quede vacía, automáticamente se borra el módulo `Diagnóstico` del "*focus stack*" y se carga el siguiente de la lista para proseguir la ejecución de las reglas que aparezcan en la agenda correspondiente.

En cualquier momento, podemos añadir un nuevo módulo a la lista, que pasará a ser el módulo activo, con

```
(focus <Nombre Módulo>)
```

Esto lo podemos hacer desde línea de comandos de CLIPS o desde alguna RHS.

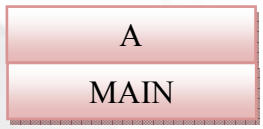
Al principio, la *"focus stack"* sólo tendrá MAIN, que siempre será el último módulo de la lista (como la función main de un programa en C). Cuando se ejecute `(focus Diagnóstico)` obtendremos una situación como la de la figura de arriba.

`ejemplo.modulos.clp`

```
(defmodule A)
…
(defmodule B)
…
```

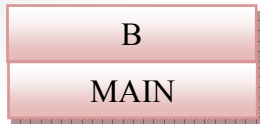
```
CLIPS> (clear)
CLIPS> (load ejemplo.modulos.clp)
CLIPS> (reset)
```


```
CLIPS> (focus A)
```




```
CLIPS> (run)
```


```
CLIPS> (focus B)
```




```
CLIPS> (run)
```

EJEMPLO

```
(defmodule M1)
      (deffacts HechosM1
            (A))
      (defrule R1
            (A)
            =>
            (assert (B)))

(defmodule M2)
      (deffacts HechosM2
            (C))
      (defrule R2
            (C)
            =>
            (assert (D)))



CLIPS> (reset)          ; Añade HechosM1 a la M.T. del módulo M1
                        ; Añade HechosM2 a la M.T. del módulo M2

CLIPS> (focus M1)       ; Se añade M1 a la 'focus stack'

CLIPS> (run)            ; Se ejecuta R1 y se añade B a la M.T. de M1

                        ; Como la agenda de M1 queda vacía,
                        ; M1 desaparece de la pila (sólo queda MAIN)

CLIPS> (focus M2)       ; Se añade M2 a la 'focus stack'

CLIPS> (run)            ; Se ejecuta R2 y se añade D a la M.T. de M2

                        ; Como la agenda de M2 queda vacía,
                        ; M2 desaparece de la pila (sólo queda MAIN)
```

EJEMPLO

```
(defmodule M1)
      (deffacts HechosM1
            (A))
      (defrule R1
            (A)
            =>
            (assert (B))
            (focus M2))

(defmodule M2)
      (deffacts HechosM2
            (C))
      (defrule R2
            (C)
            =>
            (assert (D)))
```

```
CLIPS> (reset)          ; Añade HechosM1 a la M.T. del módulo M1
                        ; Añade HechosM2 a la M.T. del módulo M2

CLIPS> (focus M1)       ; Se añade M1 a la 'focus stack'

CLIPS> (run)            ; Se ejecuta R1 y se añade B a la M.T. de M1

                        ; Se ejecuta (focus M2),
                        ; por lo que se añade M2 a la 'focus stack'

                        ; Se ejecuta R2 y se añade D a la M.T. de M2

                        ; Como la agenda de M2 queda vacía,
                        ; M2 desaparece de la pila (quedan M1 y MAIN)

                        ; Como la agenda de M1 está vacía,
                        ; M1 desaparece de la pila (sólo queda MAIN)
CLIPS>
```

Importar y exportar construcciones

Cada módulo tiene asociados hechos y reglas que son invisibles para otros módulos a no ser que se exporten. Para hacerlo, hemos de tener en cuenta que:

- No se pueden exportar hechos aislados (se exporta un `deftemplate` determinado y, automáticamente, se está exportando la definición del template junto con los hechos correspondientes).

- Un módulo no dice a quién va a exportar, pero el que importa sí debe decir de dónde lo va a hacer.

```
(defmodule MDeteccion
      (export deftemplate FichaPaciente))

(deftemplate FichaPaciente
      (field Paciente)
      (field Tipo))

(deffacts Hechos
      (FichaPaciente (Paciente Juan)
                     (Tipo     Interno)))

(defmodule MDiagnostico
      (import MDeteccion deftemplate FichaPaciente))

(defrule Regla
      (FichaPaciente (Tipo Interno)
                     (Paciente ?Nombre))
      …
```

`MDetección` exporta `FichaPaciente` y `MDiagnostico` la importa.

> NOTA: Si quisiéramos exportar los vectores ordenados de características que estén definidos dentro de un `deffacts`, debemos obligatoriamente exportar e importar TODOS los `deftemplates` definidos en el módulo.

Instrucciones de importación y exportación:

```
(export ?ALL)                    exporta todo.
(export ?NONE)                   no exporta nada (por defecto).
(export deftemplate ?ALL)        exporta todos los deftemplates y vectores.
(export deftemplate ?NONE)       no exporta ningún deftemplate.
(export deftemplate <nombre>+)   sólo exporta los deftemplates especificados.

(import <módulo> ?ALL)
(import <módulo> ?NONE)
(import <módulo> deftemplate ?ALL)
(import <módulo> deftemplate ?NONE)
(import <módulo> deftemplate <nombre>+)
```

Se pueden usar los hechos (definidos en un `deffacts`) de un módulo que los exporta, sin que tenga que pasar el foco por él.

```
(defmodule A (export ?ALL))

(deftemplate Registro (field F))

(deffacts Hechos
      (vector 1)
      (Registro (F 1)))


(defmodule B (import A ?ALL))

(defrule R
      (vector ?variable)
      (Registro (F 1))
      =>
      (printout t ?variable))
```

Obviamente, si los hechos no están definidos dentro de un `deffacts`, sino que son asertados en la RHS de alguna regla del módulo `A`, entonces el foco debe pasar antes por `A` (y la regla debe ejecutarse para que el hecho exista en la memoria de trabajo).

NOTA: No se pueden usar en el MAIN hechos asertados desde otros módulos, ni al revés (en los módulos no se conocen ni los hechos ni los deftemplates definidos en MAIN).

Control de fases mediante módulos

```
(deffacts InformacionDeControl
        (ListaFases MDeteccion MDiagnostico MTerapia))

(defrule CambiarFase
      ?Lista <- (ListaFases ?Siguiente $?Resto)
      =>
      (focus ?Siguiente)
      (retract ?Lista)
      (assert (ListaFases $?Resto ?Siguiente)))


(defmodule MDeteccion)
…

(defmodule MDiagnostico)
…

(defmodule MTerapia)
…
```

Obsérvese que ya no introducimos prioridades (`-10`) en la regla `CambiarFase`, ya que ahora no resultan necesarias.

Sin embargo, hay un problema si no existen reglas aplicables en ningún módulo, ya que al ejecutar `(run)` la regla `CambiarFase` se activa y ejecuta indefinidamente.

POSIBLE SOLUCIÓN: Incluir otro patrón en la regla de cambio de fase.

Para el ejemplo anterior, es lógico que si hemos resuelto el problema de un paciente y ya no hay más pacientes, entonces no se cambia de fase.

Su suponemos que el usuario introducirá algo del tipo:

```
CLIPS> (assert (BuscarTerapia Juan))
```

Entonces, la regla adecuada para cambiar de fase será de la forma:

```
(defrule CambiarFase
      (exists (BuscarTerapia ?))
      ?Lista <- (ListaFases ?Siguiente $?Resto)
      =>
      (focus ?Siguiente)
      (retract ?Lista)
      (assert (ListaFases $?Resto ?Siguiente)))
```