UNIVERSITAT POLITÈCNICA DE VALÈNCIA

# Guia de programación en CLIPS

## *Extractos de la guía de programación bàsica para SIN*

Albert Sanchis
Alfons Juan

DSIC

Departamento de Sistemas Informáticos y Computación

# Índice

# Prólogo: historia y documentación de CLIPS

- 1984: el grupo de IA del *NASA's Johnson Space Center* decide desarrollar una herramienta C de construcción de sistemas expertos
- 1985: se desarrolla la versión prototipo de ***C Language Integrated Production System (CLIPS)***, idónea para formación
- 1986: CLIPS se comparte con grupos externos
- 1987–2002: mejoras de rendimiento y nuevas funcionalidades; por ejemplo, programación procedural, OO e interfaces gráficas
- 2008–2020: Gary Riley mantiene CLIPS fuera de la NASA [1, 2]
- Documentación:
  - ***Manual de referència I: Guia de programación bàsica*** [3]
  - Manual de referencia II: Guía de programación avanzada [4]
  - Manual de referencia III: Guía de interfaces [5].
  - Guía del usuario [6]

# 1. Introducción

- Presentación basada en la ***Guía de programación básica*** [3]:
  - ***Misma numeración de secciones que la guía!***
  - Excluye secciones innecesarias en SIN
  - Para Linux con ejecución batch (no interactiva)
  - Sección 2: resumen de CLIPS y terminología básica
    - ↦ La sección 2.1 incluye guía rápida extra con “hola.clp”
  - Secs. 4-7: constructores deffacts, defrule, defglobal y deffunction
  - Sección 12: acciones y funciones CLIPS
  - Sección 13: ordenes típicamente interactivas

# 2. Resumen de CLIPS

## 2.1. Interacción con CLIPS y guía rápida

- *Ordenes desde la línea de órdenes:*

```
1 $ clips
2          CLIPS (6.30 3/17/15)
3 CLIPS> (+ 3 4)                          ; llamada a función
4 7
5 CLIPS> (exit)                           ; o Ctrl-C
```

- ***Entrada y carga de órdenes automática:***

  `clips [-f <f.clp>|-f2 <f.clp>|-l <f.clp>]`

  - `-f <f.clp>`: CLIPS ejecuta las órdenes del fichero `<f.clp>`
  - `-f2 <f.clp>`: Como `-f`, pero las órdenes no se muestran.
  - `-l <f.clp>`: CLIPS hace `(load <f.clp>)` inicialmente.
  - ***Gastaremos -f2 !***

- ***CLIPS*** permite construir SBRs con 3 componentes:

1. ***Base de hechos (BH):***
   - Cada estado del problema suele representarse con un único hecho de acuerdo con un cierto patrón de ***hecho-estado***
   - A cada paso de ejecución, los hecho-estado representan estados del problema ya explorados o pendientes de exploración
   - El resto de hechos son información ***estática*** del problema
2. ***Base de reglas (BR):***
   - Cada posible acción aplicable a uno o más estados del problema suele representarse con una única regla `izq=>der`
   - La parte izquierda escoge el conjunto de estados al cual es aplicable
   - La parte derecha suele resultar con nuevos hecho-estado añadidos a la BH
3. ***Motor de inferencia:*** instanciación, selección y ejecución de reglas

- ***Motor de inferencia:***
  - ***Entrada:*** base de hechos y base de reglas iniciales, $BH$ y $BR$
  - ***Salida:*** base de hechos final, $BH$
  - ***Método:***

$CC = \emptyset$ // conjunto conflicto de instancias de reglas

**repetir**

// añadimos nuevas instancias al $CC$ a partir de nuevos hechos:
$CC =$ Instancia$(BH, BR, CC)$

**si** $CC = \emptyset$: **salir** // objetivo no conseguido

// seleccionamos una instancia con algún criterio:
$InstRule =$ Selecciona$(CC)$

// ejecutamos $InstRule$ y actualizamos $BH$ y $CC$:
$(BF, CC) =$ Ejecuta$(BF, CC, InstRule)$

**hasta** objectivo conseguido

- ***Tres pasos básicos en inferencia:***

1. ***Instancia:*** añade nuevas instancias al $CC$ a partir de nuevos hechos, sin repetir instancias añadidas anteriormente ***(refracción)***
2. ***Selecciona:*** aplica un criterio de selección como ahora:
   - ***Profundidad:*** primero la instancia más reciente
   - ***Anchura:*** primero la instancia más antigua
   - ***Prioridad:*** primero la instancia de la regla más prioritaria
3. ***Ejecuta:*** aplica las órdenes de la instancia seleccionada:
   - ***Eliminación de hechos*** en la BH
   - ***Eliminación de instancias*** en el CC con hechos eliminados
   - ***Inserción de hechos*** nuevos en la BH ***sin repeticiones***

## ▸ *Un SBR sencillo: hola.clp*

2.1.hola.clp

```
1  (deffacts bf (pendiente Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendiente ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6    (retract ?f)
7    (assert (pendiente $?y)))
8  (defrule acaba (pendiente) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

clips -f2 2.1.hola.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (pendiente Manel Nora Laia)
3  ==> Activation 0      saluda: f-1
4  Hola Manel
5  <== f-1     (pendiente Manel Nora Laia)
6  ==> f-2     (pendiente Nora Laia)
7  ==> Activation 0      saluda: f-2
8  Hola Nora
9  <== f-2     (pendiente Nora Laia)
10 ==> f-3     (pendiente Laia)
11 ==> Activation 0      saluda: f-3
12 Hola Laia
13 <== f-3     (pendiente Laia)
14 ==> f-4     (pendiente)
15 ==> Activation 0      acaba: f-4
```

## ▸ hola.clp sin eliminación de hechos: hola2.clp

2.1.hola2.clp

```
1  (deffacts bf (pendiente Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendiente ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6  ;;  (retract ?f)
7    (assert (pendiente $?y)))
8  (defrule acaba (pendiente) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

clips -f2 2.1.hola2.clp

```
1  <== f-0     (initial-fact)
2  ==> f-0     (initial-fact)
3  ==> f-1     (pendiente Manel Nora Laia)
4  ==> Activation 0      saluda: f-1
5  Hola Manel
6  ==> f-2     (pendiente Nora Laia)
7  ==> Activation 0      saluda: f-2
8  Hola Nora
9  ==> f-3     (pendiente Laia)
10 ==> Activation 0      saluda: f-3
11 Hola Laia
12 ==> f-4     (pendiente)
13 ==> Activation 0      acaba: f-4
```

# 2.3. Elementos de programación básicos

## 2.3.1. Tipos de datos

- ***número:*** ***entero (integer)*** y ***real (float)***
  `237 +12 15.09 -32.3e-7`
- ***símbolo:*** secuencia de carácteres imprimibles hasta ***delimitador***
  `foo bad_value 127A 456-93-039 @+=-\%`
- ***cadena:*** `"foo" "a and b" "1 number"`
- ***hecho:*** lista de valores atómicos referenciados por posición o nombre;
  *dirección de hecho:* `<Fact-XXX>` donde `XXX` es el índice del hecho
- ***valor:*** ***único*** (campo) o ***multicampo***
  `(a) (1 bar foo) () (x 3.0 "red" 567)`

## 2.3.2. Funciones

- Código con nombre que devuelve un valor (***función***) o no (***orden***):
  - ***Definidas por el usuario en CLIPS:*** `deffunction`
  - ***Predefinidas [3, ap. H]:***
    `!= * ** + - / < <= <> = >= >= abs acos ...`
  - ***Llamadas:*** en notación prefija, `(+ 3 4 5)`

## 2.3.3. Constructores

- `defglobal:` definición de variables globales
- `deffacts:` hechos automáticamente insertados con `reset`
- `deffunction:` funciones definidas por el usuario
- `defrule:` definición de reglas

# 2.4. Abstracción de datos

## 2.4.1. Hechos

- ***Ordenados:*** lista de símbolos entre paréntesis donde el primero (distinto de `test`, `and`, etc.) indica la "relación" `(compra ajo sal)`
- ***Órdenes*** `assert retract`
- La inserción de un hecho repetido no tiene efecto (se ignora)
- El índice o dirección de un hecho puede obtenerse en la parte izquierda de una regla o como valor devuelto de `assert`
- ***Hechos iniciales:*** con `deffacts`

## 2.4.3. Variables globales

Se definen con `defglobal`

# 2.5. Representación del conocimiento

## 2.5.1. Conocimiento heurístico: reglas

- ***Reglas:*** constan de dos partes, la LHS y la RHS
  - ***Antecedente o parte izquieda (LHS):***
    - ↦ Condiciones a cumplir para que se ejecute la RHS
    - ↦ ***Patrón:*** tipos de condición muy importante
  - ***Consecuente o parte derecha (RHS):***
    - ↦ Acciones a ejecutar si se cumple la LHS
- ***Motor de inferencia:*** hace el ***pattern matching*** (encaje de patrones)
- ***Estrategia de resolución de conflictos:*** decide qué regla ejecuta si hay más de una aplicable

## 2.5.2. Conocimiento procedimental

- *Código como el de los lenguages convencionales:*
  - *Funciones definidas por el usuario en CLIPS:* `deffunction`

# 4. Constructor deffacts

- ***deffacts*** inserta (o reconstruye) la lista de hechos con **`reset`**

```
(deffacts <deffacts-name> [<comment>] <RHS-pattern>*)
```

- *Ejemplo:*

4.deffacts.clp

```
(deffacts bf (pendiente Manel Nora Laia))
(watch facts)
(reset)
(exit)
```

clips -f2 4.deffacts.clp

```
<== f-0     (initial-fact)
==> f-0     (initial-fact)
==> f-1     (pendiente Manel Nora Laia)
```

# 5. Constructor defrule

- ***Regla:***
  - Condiciones y acciones a ejecutar si las condiciones se cumplen
  - Se ejecuta o dispara (*fire*) en función de la existencia o no de hechos con los cuales se cumplen las condiciones
  - El motor de inferencia es el encargado de encajar (hacer *matching* de) hechos con reglas
    - ↦ Llamamos ***instancias*** de una regla a los diferentes matchings de hechos con la regla que se puedan hacer (cero, uno o más)
    - ↦ ***Agenda o conjunto conflicto:*** conjunto de instancias de todas las reglas pendientes de ejecución

# 5.1. Definiendo reglas

- ***defrule*** define una regla con:

```
1 (defrule <rule-name> [<comment>]
2   [<declaration>]                ; Rule Properties
3   <conditional-element>*         ; Left-Hand Side (LHS)
4   =>
5   <action>*)                     ; Right-Hand Side (RHS)
```

- *Ejemplo:*

5.1.defrule.clp

```
1 (defrule izquierda
2   (robot ?x ?y)
3   =>
4   (assert (robot (- ?x 1) ?y)))
```

# 5.2. Ciclo básico de ejecución de reglas

- ***Motor de inferencia:*** bucle con los siguientes pasos básicos

a) ***Selección de una instancia (de regla) de la agenda***

- Si no hay ninguna instancia en la agenda, se acaba

b) ***Ejecución de la parte derecha de la regla seleccionada***

c) ***Activación y desactivación de instancias de reglas*** como consecuencia de la ejecución de la regla seleccionada

- Las activadas se añaden a la agenda
- Las desactivadas se eliminan de la agenda

d) ***Re-evaluación de prioridades dinámicas de instancias en la agenda:*** si utilizamos prioridades dinámicas con `salience`

# 5.3. Estrategias de resolución de conflictos

- ***Ordenación de instancias de reglas en la agenda:***
  - ***Por prioridad:*** las nuevas instancias se situan encima (delante) de las de menor prioridad y bajo (detrás) de las de mayor
  - ***En caso de empate a prioridad:*** aplicamos la ***estrategia de resolución de conflictos***, como ahora "las más nuevas delante"
    - ↦ En caso de empate a prioridad y con dos o más instancias activadas al mismo tiempo (para una misma inserción o borrado de un hecho), puede ser no podemos ordenarlas con dicha estrategia
      - Entonces, se ordenan arbitrariamente (*no aleatoriamente*), en general de acuerdo con el orden de definición de las reglas, como ahora "las definidas más nuevas (últimas definidas) delante"

### 5.3.1. Profundidad (*depth*)

- Las nuevas instancias se sitúan encima de todas las de igual prioridad; es la estrategia per defecto

### 5.3.2. Anchura (*breadth*)

- Las nuevas reglas se sitúan bajo de todas las de igual prioridad

# 5.4. Sintaxis de la LHS

- ***Elementos condicionales (CEs):*** serie de cero, uno o más elementos de que consta la LHS de una regla y que se han de satisfacer para que se añada una instancia de la regla a la agenda

- Hay ocho tipos de CEs, pero sólo hacemos uso de cinco:
  - ***CEs patrón:*** restricciones sobre los hechos que lo satisfacen
  - ***CEs test:*** evalúan expresiones durante el encaje de patrones
  - ***CEs or:*** dado un grupo de CEs, al menos uno se ha de satisfacer
  - ***CEs and:*** dado un grupo de CEs, todos se han de satisfacer
  - ***CEs not:*** dado un CE, *no* se ha de satisfacer

```
1  <conditional-element> ::=
2    <pattern-CE> | <assigned-pattern-CE> |
3    <test-CE> | <or-CE> | <and-CE> | <not-CE>
```

## 5.4.1. CE patrón

- ***CE patrón:*** lista ordenada con un símbolo inicial, seguido de constantes, comodines y variables, puede ser precedida de una dirección de patrón
  - ***Restricciones literales:*** constantes
  - ***Comodines:***
    - ***Mono-valuados:*** `?`
    - ***Multi-valuados:*** `$?`
  - ***Variables:***
    - ***Mono-valuadas:*** `?<var>`
    - ***Multi-valuadas:*** `$?<var>`
  - ***Restricciones de valor de retorno:*** `=<func>`
  - ***Direciones de patrón:*** `?<var> <- <CE patron>`

## ▸ *Restricciones literales:* constantes

5.4.1.literales.clp

```
1  (deffacts data-facts
2  (data 1.0 blue "red")
3  (data 1 blue)
4  (data 1 blue red)
5  (data 1 blue RED)
6  (data 1 blue red 6.9))
7  (defrule find-data (data 1 blue red) =>)
8  (watch facts)
9  (watch activations)
10 (reset)
11 (exit)
```

clips -f2 5.4.1.literales.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (data 1.0 blue "red")
3 ==> f-2     (data 1 blue)
4 ==> f-3     (data 1 blue red)
5 ==> Activation 0      find-data: f-3
6 ==> f-4     (data 1 blue RED)
7 ==> f-5     (data 1 blue red 6.9)
```

- ***Comodines mono-valuados y multi-valuados:*** `?` encaja con un campo exactamente y `$?` con cero o más

5.4.1.comodines.clp

```
1  (deffacts data-facts
2  (data 1.0 blue "red")
3  (data 1 blue)
4  (data 1 blue red)
5  (data 1 blue RED)
6  (data 1 blue red 6.9))
7  (defrule find-data (data ? blue red $?) =>)
8  (watch facts)
9  (watch activations)
10 (reset)
11 (exit)
```

clips -f2 5.4.1.comodines.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (data 1.0 blue "red")
3  ==> f-2     (data 1 blue)
4  ==> f-3     (data 1 blue red)
5  ==> Activation 0      find-data: f-3
6  ==> f-4     (data 1 blue RED)
7  ==> f-5     (data 1 blue red 6.9)
8  ==> Activation 0      find-data: f-5
```

- ***Variables mono-valuadas y multi-valuadas:*** `?<var>` encaja con un campo exactamente y `$?<var>` con cero o más

5.4.1.variables.clp

```
1  (deffacts data-facts (data 1 blue) (data 1 blue red)
2    (data 1 blue red 6.9))
3  (defrule find-data-1
4    (data ?x $?y ?z)
5    => (printout t "?x=" ?x " $?y=" $?y " ?z=" ?z crlf ))
6  (watch facts)
7  (watch activations)
8  (set-strategy breadth) ; por omisión es depth
9  (reset)
10 (run)
11 (exit)
```

clips -f2 5.4.1.variables.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (data 1 blue)
3  ==> Activation 0      find-data-1: f-1
4  ==> f-2     (data 1 blue red)
5  ==> Activation 0      find-data-1: f-2
6  ==> f-3     (data 1 blue red 6.9)
7  ==> Activation 0      find-data-1: f-3
8  ?x=1 $?y=() ?z=blue
9  ?x=1 $?y=(blue) ?z=red
10 ?x=1 $?y=(blue red) ?z=6.9
```

## ▸ Restricciones de valor de retorno: =<func>

5.4.1.retorno.clp

```
1 (deffacts bf (xy 1 2) (xy 3 3) (xy 4 8) (xy 5 8))
2 (defrule encuentra-doble (xy ?x =(* 2 ?x)) =>
3   (printout t "?x=" ?x crlf))
4 (watch facts)
5 (watch activations)
6 (reset)
7 (run)
8 (exit)
```

clips -f2 5.4.1.retorno.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (xy 1 2)
3 ==> Activation 0      encuentra-doble: f-1
4 ==> f-2     (xy 3 3)
5 ==> f-3     (xy 4 8)
6 ==> Activation 0      encuentra-doble: f-3
7 ==> f-4     (xy 5 8)
8 ?x=4
9 ?x=1
```

## ▸ *Direcciones de patrón:* `?<var> <- <pattern-CE>`

5.4.1.direcciones.clp

```
1 (deffacts bf (color rojo) (color verde))
2 (defrule encuentra-color
3   ?f <- (color ?c)
4   => (printout t ?c " encontrado en el hecho " ?f crlf))
5 (watch facts)
6 (watch activations)
7 (reset)
8 (run)
9 (exit)
```

clips -f2 5.4.1.direcciones.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (color rojo)
3 ==> Activation 0      encuentra-color: f-1
4 ==> f-2     (color verde)
5 ==> Activation 0      encuentra-color: f-2
6 verde encontrado en el hecho <Fact-2>
7 rojo encontrado en el hecho <Fact-1>
```

# 5.4.2. CE test

- `(test <func>)` se satisface si `<func>` no devuelve falso

5.4.2.test.clp

```
1  (deffacts bf (tenemos 6 platos) (tenemos 5 vasos))
2  (defrule tenemos-mas
3    (tenemos ?n ?x)
4    (tenemos ?m ?y)
5    (test (> ?n ?m))
6    =>
7    (printout t "Tenemos más " ?x " que " ?y crlf))
8  (watch facts)
9  (watch activations)
10 (reset)
11 (run)
12 (exit)
```

clips -f2 5.4.2.test.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (tenemos 6 platos)
3 ==> f-2     (tenemos 5 vasos)
4 ==> Activation 0      tenemos-mas: f-1,f-2
5 Tenemos más platos que vasos
```

## 5.4.3. CE or

- `(or <CE>+)` se satisface si cualquier de los `<CE>+` lo hace

5.4.3.or.clp

```
1  (deffacts bf (obstaculo 5 3) (robot 4 3))
2  (defrule obstaculo-al-lado
3    (robot ?x ?y)
4    (or (obstaculo =(- ?x 1) ?y) (obstaculo =(+ ?x 1) ?y))
5    =>
6    (printout t "Tenemos obstaculo al lado" crlf))
7  (watch facts)
8  (watch activations)
9  (reset)
10 (run)
11 (exit)
```

clips -f2 5.4.3.or.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (obstaculo 5 3)
3  ==> f-2     (robot 4 3)
4  ==> Activation 0      obstaculo-al-lado: f-2,f-1
5  Tenemos obstaculo al lado
```

# 5.4.4. CE and

- (and <CE>+) se satisface si todos los <CE>+ lo hacen

5.4.4.and.clp

```
1  (deffacts bf (obstaculo 3 3) (obstaculo 5 3) (robot 4 3))
2  (defrule bloqueado-por-los-lados
3    (robot ?x ?y)
4    (and (obstaculo =(- ?x 1) ?y) (obstaculo =(+ ?x 1) ?y))
5    =>
6    (printout t "Robot bloqueado por los lados" crlf))
7  (watch facts)
8  (watch activations)
9  (reset)
10 (run)
11 (exit)
```

clips -f2 5.4.4.and.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (obstaculo 3 3)
3  ==> f-2     (obstaculo 5 3)
4  ==> f-3     (robot 4 3)
5  ==> Activation 0      bloqueado-por-los-lados: f-3,f-1,f-2
6  Robot bloqueado por los lados
```

# 5.4.5. CE not

- `(not <CE>+)` se satisface si `<CE>` no lo hace

5.4.5.not.clp

```
1 (deffacts bf (obstaculo 1 3) (robot 4 3))
2 (defrule izquierda
3   (robot ?x ?y) (not (obstaculo =(- ?x 1) ?y))
4   => (assert (robot (- ?x 1) ?y)))
5 (watch facts)
6 (watch activations)
7 (reset)
8 (run)
9 (exit)
```

clips -f2 5.4.5.not.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (obstaculo 1 3)
3 ==> f-2     (robot 4 3)
4 ==> Activation 0      izquierda: f-2,
5 ==> f-3     (robot 3 3)
6 ==> Activation 0      izquierda: f-3,
7 ==> f-4     (robot 2 3)
```

## 5.4.10. Declaración de propiedades de regla

- Con `(salience <entero>)` definimos la prioridad de la regla
  - ***Prioridad mínima:*** -10 000
  - ***Prioridad por defecto:*** 0
  - ***Prioridad máxima:*** 10 000

**5.4.10.salience.clp**

```
1 (deffacts bf (hecho-estado-obj a b))
2 (defrule obj
3   (declare (salience 1)) ; entre -10000 y +10000; 0 por defecto
4   (hecho-estado-obj a b)
5   =>
6   (printout t "Solución encontrada!" crlf))
7 (reset)
8 (run)
9 (exit)
```

**clips -f2 5.4.10.salience.clp**

```
1 Solución encontrada!
```

# 6. Constructor defglobal

- `defglobal` define una ***variable global*** y le da valor:

```
1 (defglobal [<defmodule-name>] <global-assignment>*)
2 <global-assignment> ::= <global-variable> = <expression>
3 <global-variable>   ::= ?*<symbol>*
```

- ***Peligro de uso inapropiado por programadores inexpertos:***
  - A diferencia de la inserción o borrado de hechos, la modificación de variables globales no provoca la activación o desactivación de instancias de reglas; son memoria al margen de la búsqueda en árbol
  - Las utilizamos para contar nodos (hechos del árbol de búsqueda) insertados o limitar la profundidad del árbol de búsqueda; poco más

6.defglobal.clp

```
1  (defglobal ?*N* = 0)
2  (deffacts bf (L a b a b a))
3  (defrule R
4    ?f <-(L ?x $?y ?x $?z)
5    =>
6    (printout t ?x" "?y" "?z
     ↪  crlf)
7    (retract ?f)
8    (assert (L $?y ?x $?z))
9    (bind ?*N* (+ ?*N* 1)))
10 (watch facts)
11 (watch activations)
12 (watch globals)
13 (set-strategy breadth)
14 (reset)
15 (run)
16 (printout t "N=" ?*N* crlf)
17 (exit)
```

clips -f2 6.defglobal.clp

```
1  :== ?*N* ==> 0 <== 0
2  ==> f-0     (initial-fact)
3  ==> f-1     (L a b a b a)
4  ==> Activation 0      R: f-1
5  ==> Activation 0      R: f-1
6  a (b a b) ()
7  <== f-1     (L a b a b a)
8  <== Activation 0      R: f-1
9  ==> f-2     (L b a b a)
10 ==> Activation 0      R: f-2
11 :== ?*N* ==> 1 <== 0
12 b (a) (a)
13 <== f-2     (L b a b a)
14 ==> f-3     (L a b a)
15 ==> Activation 0      R: f-3
16 :== ?*N* ==> 2 <== 1
17 a (b) ()
18 <== f-3     (L a b a)
19 ==> f-4     (L b a)
20 :== ?*N* ==> 3 <== 2
21 N=3
```

# 7. Constructor deffunction

- `deffunction` define funciones de usuario

```
1 (deffunction <name> [<comment>]
2   (<regular-parameter>* [<wildcard-parameter>])
3   <action>*)
4 <regular-parameter>  ::= <single-field-variable>
5 <wildcard-parameter> ::= <multifield-variable>
```

  - ***Cero o más variables mono-valuadas:*** en primer lugar tiene cero o más parámetros convencionales, todos ellos variables mono-valuadas, por lo que hemos de pasarle tantos argumentos como variables mono-valuadas tenga

  - ***Seguidas de una variable multi-valuada opcional:*** en último lugar tiene una variable multi-valuada opcional; si la tiene, podemos pasarle tantos argumentos adicionales como queramos

7.deffunction.clp

```
1 (deffunction print-args (?a ?b $?c)
2   (printout t ?a " " ?b " and " (length ?c) " extras: " ?c
    ↪  crlf))
3 (print-args 1 2)
4 (print-args a b c d)
5 (exit)
```

clips -f2 7.deffunction.clp

```
1 1 2 and 0 extras: ()
2 a b and 2 extras: (c d)
```

# 12. Acciones y funciones

- ***Acciones, funciones y órdenes:*** en realidad son todo funciones que se pueden utilizar en reglas, funciones definidas por el usuario o en la línea de órdenes

- Escogemos uno u otro nombre nada más por matices:

  - ***Función:*** suele referirse a una función que devuelve un valor

  - ***Acción:*** función que no devuelve ningún valor pero que realiza alguna operación básica como efecto secundario (`printout`)

  - ***Orden:*** función que suele utilizarse en la línea de órdenes y no devuelve ningún valor (`reset`) o puede ser sí (`set-strategy`)

# 12.1. Funciones predicado

12.1.pruebas.clp

```
1  (numberp 23)                    ; prueba de número entero/real
2  (floatp 3.0)                    ; prueba de real
3  (integerp 3)                    ; prueba de entero
4  (lexemep SIN)                   ; prueba de cadena o símbolo
5  (stringp "SIN")                 ; prueba de cadena
6  (symbolp SIN)                   ; prueba de símbolo
7  (evenp 2)                       ; prueba de número par
8  (oddp 3)                        ; prueba de número impar
9  (multifieldp (create$ a b))     ; prueba de multicampo
10 (exit)
```

clips -f 12.1.pruebas.clp

```
1  CLIPS> (numberp 23)                    ; prueba de número entero/real
2  TRUE
3  CLIPS> (floatp 3.0)                    ; prueba de real
4  TRUE
5  CLIPS> (integerp 3)                    ; prueba de entero
6  TRUE
7  CLIPS> (lexemep SIN)                   ; prueba de cadena o símbolo
8  TRUE
9  CLIPS> (stringp "SIN")                 ; prueba de cadena
10 TRUE
11 CLIPS> (symbolp SIN)                   ; prueba de símbolo
12 TRUE
13 CLIPS> (evenp 2)                       ; prueba de número par
14 TRUE
15 CLIPS> (oddp 3)                        ; prueba de número impar
16 TRUE
17 CLIPS> (multifieldp (create$ a b))     ; prueba de multicampo
18 TRUE
19 CLIPS> (exit)
```

12.1.comparaciones.clp

```
(eq foo foo foo foo)    ; TRUE si 1r arg igual al resto
(neq foo bar yak bar)   ; TRUE si 1r arg distinto al resto
(= 3 3.0)               ; TRUE si 1r número igual al resto
(<> 4 4.1)              ; TRUE si 1r núm. distinto al resto
(> 5 4 3)               ; TRUE si args en orden decreciente
(>= 5 5 3)              ; TRUE si args en ord. no creciente
(< 3 4 5)               ; TRUE si args en orden creciente
(<= 3 5 5)              ; TRUE si args en orden no decrec.
(exit)
```

clips -f 12.1.comparaciones.clp

```
CLIPS> (eq foo foo foo foo)    ; TRUE si 1r arg igual al resto
TRUE
CLIPS> (neq foo bar yak bar)   ; TRUE si 1r arg distinto al resto
TRUE
CLIPS> (= 3 3.0)               ; TRUE si 1r número igual al resto
TRUE
CLIPS> (<> 4 4.1)              ; TRUE si 1r núm. distinto al resto
TRUE
CLIPS> (> 5 4 3)               ; TRUE si args en orden decreciente
TRUE
CLIPS> (>= 5 5 3)              ; TRUE si args en ord. no creciente
TRUE
CLIPS> (< 3 4 5)               ; TRUE si args en orden creciente
TRUE
CLIPS> (<= 3 5 5)              ; TRUE si args en orden no decrec.
TRUE
CLIPS> (exit)
```

12.1.logicas.clp

```
1 (and TRUE (> 2 1))      ; TRUE si todos los args son TRUE
2 (or FALSE (> 2 1))      ; TRUE si cualquier arg es TRUE
3 (not (evenp 3))         ; TRUE si arg falso
4 (exit)
```

clips -f 12.1.logicas.clp

```
1 CLIPS> (and TRUE (> 2 1))      ; TRUE si todos los args son TRUE
2 TRUE
3 CLIPS> (or FALSE (> 2 1))      ; TRUE si cualquier arg es TRUE
4 TRUE
5 CLIPS> (not (evenp 3))         ; TRUE si arg falso
6 TRUE
```

# 12.2. Funciones multicampo

12.2.clp

```
1  (create$ a b)                      ; crea valor multicampo
2  (nth$ 2 (create$ a b))             ; n-ésimo campo del multicampo
3  (member$ b (create$ a b b))        ; posicion(es) de valor en mcamp
4  (member$ (create$ b b) (create$ a b b))
5  (member$ c (create$ a b b))
6  (subsetp (create$ b a) (create$ a b b)) ; mcamp1 en mcamp2?
7  (subsetp (create$ A) (create$ a b b))
8  (delete$ (create$ a b b) 2 3) ; borra mcamp de pos1 a pos2
9  (explode$ "a b")                   ; explota cadena a mcamp
10 (implode$ (create$ a b))           ; implota mcamp a cadena
11 (subseq$ (create$ a b b) 2 3) ; extrae mcamp de p1 a p2
12 (replace$ (create$ a b b) 2 3 B) ; subs mcamp-p1-p2 por valor
13 (insert$ (create$ a b b) 2 B)     ; ins en mcap-pos valor
14 (first$ (create$ a b c))           ; 1r campo de mcamp
15 (rest$ (create$ a b c))            ; resto de mcamp (= borra 1r)
16 (length$ (create$ a b c))          ; longitud de mcamp
17 (delete-member$ (create$ a b a c) b a) ; borra valores d mcamp
18 (delete-member$ (create$ a b a c b a) (create$ b a))
19 (replace-member$ (create$ a x a y) z x y) ; subs v2- x v1
20 (exit)
```

```
CLIPS> (create$ a b)                    ; crea valor multicampo
(a b)
CLIPS> (nth$ 2 (create$ a b))           ; n-ésimo campo del multicampo
b
CLIPS> (member$ b (create$ a b b))      ; posicion(es) de valor en mcamp
2
CLIPS> (member$ (create$ b b) (create$ a b b))
(2 3)
CLIPS> (member$ c (create$ a b b))
FALSE
CLIPS> (subsetp (create$ b a) (create$ a b b)) ; mcamp1 en mcamp2?
TRUE
CLIPS> (subsetp (create$ A) (create$ a b b))
FALSE
CLIPS> (delete$ (create$ a b b) 2 3) ; borra mcamp de pos1 a pos2
(a)
CLIPS> (explode$ "a b")                 ; explota cadena a mcamp
(a b)
CLIPS> (implode$ (create$ a b))         ; implota mcamp a cadena
"a b"
CLIPS> (subseq$ (create$ a b b) 2 3) ; extrae mcamp de p1 a p2
(b b)
CLIPS> (replace$ (create$ a b b) 2 3 B) ; subs mcamp-p1-p2 por valor
(a B)
CLIPS> (insert$ (create$ a b b) 2 B)    ; ins en mcap-pos valor
(a B b b)
CLIPS> (first$ (create$ a b c))         ; 1r campo de mcamp
(a)
CLIPS> (rest$ (create$ a b c))          ; resto de mcamp (= borra 1r)
(b c)
CLIPS> (length$ (create$ a b c))        ; longitud de mcamp
3
CLIPS> (delete-member$ (create$ a b a c) b a) ; borra valores d mcamp
(c)
CLIPS> (delete-member$ (create$ a b a c b a) (create$ b a))
(a c)
CLIPS> (replace-member$ (create$ a x a y) z x y) ; subs v2- x v1
(a z a z)
```

# 12.3. Funciones para cadenas

12.3.clp

```
1  (str-cat "cad" 1 sim 3.1)   ; crea cadena por concatenación
2  (sym-cat "cad" 1 sim 3.1)   ; crea símbolo por concatenación
3  (sub-string 2 3 "abc")      ; extrae subcadena entre posiciones
4  (str-index "bc" "abcbc")    ; índice de cad1 en cad2 (1a ocur.)
5  (eval "(+ 3 4)")            ; evalúa cad como una función
6  (build "(defrule R (a)=>(assert(b)))") ; evalúa constructor
7  (rules)
8  (lowcase "HolA")
9  (str-compare "cad" "cad")    ; compara cads i sims (0 si =)
10 (str-compare "cada" "cadb") ; -1 si la 1a es menor
11 (str-compare "cadb" "cada") ;  1 si la 1a es mayor
12 (str-length "abcd")         ; longitud de cadena o símbolo
13 (check-syntax "(defrule R =>)") ; comprueba sintaxis; FALSE=ok
14 (string-to-field "3.4") ; conversión de cad/sim a tipo básico
15 (exit)
```

clips -f 12.3.clp

```
CLIPS> (str-cat "cad" 1 sim 3.1)  ; crea cadena por concatenación
"cad1sim3.1"
CLIPS> (sym-cat "cad" 1 sim 3.1)  ; crea símbolo por concatenación
cad1sim3.1
CLIPS> (sub-string 2 3 "abc")     ; extrae subcadena entre posiciones
"bc"
CLIPS> (str-index "bc" "abcbc")   ; índice de cad1 en cad2 (1a ocur.)
2
CLIPS> (eval "(+ 3 4)")           ; evalúa cad como una función
7
CLIPS> (build "(defrule R (a)=>(assert(b)))") ; evalúa constructor
TRUE
CLIPS> (rules)
R
For a total of 1 defrule.
CLIPS> (lowcase "HolA")
"hola"
CLIPS> (str-compare "cad" "cad")   ; compara cads i sims (0 si =)
0
CLIPS> (str-compare "cada" "cadb") ; -1 si la 1a es menor
-1
CLIPS> (str-compare "cadb" "cada") ;  1 si la 1a es mayor
1
CLIPS> (str-length "abcd")         ; longitud de cadena o símbolo
4
CLIPS> (check-syntax "(defrule R =>)") ; comprueba sintaxis; FALSE=ok
FALSE
CLIPS> (string-to-field "3.4") ; conversión de cad/sim a tipo básico
3.4
```

# 12.4. Sistema de entrada/salida

- *Nombres lógicos:* `stdin stdout ...`

12.4.clp

```
(printout t ":)" crlf)   ; (printout <nomlogico> <expresion>*)
(open "xy" f "w")        ; (open <nomf> <nomlogico> [<mode>])
(printout f "x y" crlf)
(close f)                ; (close [<nomlogico>])
(system "cat xy")
(open "xy" f)            ; modo por omisión: "r"
(read f)
(read f)
(read f)
(close f)
(open "xy" f)
(readline f)
(close)
; ... format rename remove get-char read-number set-locale
(exit)
```

clips -f 12.4.clp

```
1  CLIPS> (printout t ":)" crlf)   ; (printout <nomlogico> <expresion>*)
2  :)
3  CLIPS> (open "xy" f "w")        ; (open <nomf> <nomlogico> [<mode>])
4  TRUE
5  CLIPS> (printout f "x y" crlf)
6  CLIPS> (close f)                ; (close [<nomlogico>])
7  TRUE
8  CLIPS> (system "cat xy")
9  x y
10 CLIPS> (open "xy" f)            ; modo por omisión: "r"
11 TRUE
12 CLIPS> (read f)
13 x
14 CLIPS> (read f)
15 y
16 CLIPS> (read f)
17 EOF
18 CLIPS> (close f)
19 TRUE
20 CLIPS> (open "xy" f)
21 TRUE
22 CLIPS> (readline f)
23 "x y"
24 CLIPS> (close)
25 TRUE
26 CLIPS> ; ... format rename remove get-char read-number set-locale
```

# 12.5. Funciones matemáticas

12.5.clp

```
1  (+ 2 3 4)                    ; suma
2  (- 12 3 4)                   ; resta
3  (* 2 3 4)                    ; multiplicación
4  (/ 24 3 4)                   ; división
5  (div 5 2)                    ; división entera
6  (max 3.0 4 2.0)              ; máximo numérico
7  (min 4 0.1 -2.3)             ; mínimo numérico
8  (abs -2)                     ; valor absoluto
9  (float -2)                   ; conversión a real
10 (integer 4.0)                ; conversión a entero
11 (cos 0)                      ; coseno (cosh sin sinh tan  ...)
12 (acos 1.0)                   ; arcocoseno (acosh asin asinh ...)
13 (deg-grad 90)                ; grados: deg-rad grad-deg rad-deg pi
14 (sqrt 9)                     ; raiz cuadrada
15 (** 3 2)                     ; exponenciación
16 (exp 1)                      ; exponenciación natural
17 (log 2.71828182845905)       ; logaritmo natural, log10 decimal
18 (round 3.6)                  ; redondeo al entero más próximo
19 (mod 5 2)                    ; resta
20 (exit)
```

```
CLIPS> (+ 2 3 4)                ; suma
9
CLIPS> (- 12 3 4)               ; resta
5
CLIPS> (* 2 3 4)                ; multiplicación
24
CLIPS> (/ 24 3 4)               ; división
2.0
CLIPS> (div 5 2)                ; división entera
2
CLIPS> (max 3.0 4 2.0)          ; máximo numérico
4
CLIPS> (min 4 0.1 -2.3)         ; mínimo numérico
-2.3
CLIPS> (abs -2)                 ; valor absoluto
2
CLIPS> (float -2)               ; conversión a real
-2.0
CLIPS> (integer 4.0)            ; conversión a entero
4
CLIPS> (cos 0)                  ; coseno (cosh sin sinh tan  ...)
1.0
CLIPS> (acos 1.0)               ; arcocoseno (acosh asin asinh ...)
0.0
CLIPS> (deg-grad 90)            ; grados: deg-rad grad-deg rad-deg pi
100.0
CLIPS> (sqrt 9)                 ; raiz cuadrada
3.0
CLIPS> (** 3 2)                 ; exponenciación
9.0
CLIPS> (exp 1)                  ; exponenciación natural
2.71828182845905
CLIPS> (log 2.71828182845905)   ; logaritmo natural, log10 decimal
1.0
CLIPS> (round 3.6)              ; redondeo al entero más próximo
4
CLIPS> (mod 5 2)                ; resta
1
```

# 12.6. Funciones procedimentales

- `bind` asigna valores a variables:

12.6.bind.clp

```
(defglobal ?*x* = 3.4)   ; def vble global y le da valor
?*x*
(bind ?*x* (+ 8 9))      ; modif valor vble global
?*x*
(bind ?a 3)              ; crea vble local y le da valor
?a                       ; necesario CLIPS v6.30+
(deffunction f() (bind ?a 3) (bind ?a (- ?a 1)) ?a)
(f)
(exit)
```

clips -f 12.6.bind.clp

```
CLIPS> (defglobal ?*x* = 3.4)   ; def vble global y le da valor
CLIPS> ?*x*
3.4
CLIPS> (bind ?*x* (+ 8 9))      ; modif valor vble global
17
CLIPS> ?*x*
17
CLIPS> (bind ?a 3)              ; crea vble local y le da valor
3
CLIPS> ?a                       ; necesario CLIPS v6.30+
3
CLIPS> (deffunction f() (bind ?a 3) (bind ?a (- ?a 1)) ?a)
CLIPS> (f)
2
CLIPS> (exit)
```

- (if <exp> then <action>* [else <action>*]):

12.6.ifthenelse.clp

```
1  (defglobal ?*prof* = 60)
2  (deffunction inicio () ; para sistemas CLIPS interactivos
3    (reset)
4    (printout t "Profundidad maxima: ")
5    (bind ?*prof* (read))
6    (printout t "Anchura (1) o Profundidad (2): ")
7    (bind ?a (read))
8    (if (= ?a 1)
9      then (set-strategy breadth)
10     else (set-strategy depth)))
```

clips interactiu

```
1  CLIPS> (load "12.6.ifthenelse.clp")
2  Defining defglobal: prof
3  Defining deffunction: inici
4  TRUE
5  CLIPS> (inici)
6  Profunditat maxima: 50
7  Amplaria (1) o profunditat (2): 1
8  depth
9  CLIPS> (exit)
```

- (while <expression> [do] <action>*):

12.6.while.clp

```
1 (deffunction bucle (?n) ; FALSE si no acaba con return
2   (bind ?i 0)
3   (while (< ?i ?n) (printout t ?i crlf) (bind ?i (+ ?i 1))))
4 (bucle 4)
5 (exit)
```

clips -f 12.6.while.clp

```
1 CLIPS> (deffunction bucle (?n) ; FALSE si no acaba con return
2   (bind ?i 0)
3   (while (< ?i ?n) (printout t ?i crlf) (bind ?i (+ ?i 1))))
4 CLIPS> (bucle 4)
5 0
6 1
7 2
8 3
9 FALSE
```

- `(loop-for-count <range-spec> [do] <action>*)`

  `<range-spec> ::= (<loop-var> <start> <end>)`:

12.6.loop-for-count.clp

```
1 (loop-for-count (?i 0 3) (printout t ?i crlf))
2 (exit)
```

clips -f 12.6.loop-for-count.clp

```
1 CLIPS> (loop-for-count (?i 0 3) (printout t ?i crlf))
2 0
3 1
4 2
5 3
6 FALSE
7 CLIPS> (exit)
```

- `(return [<expression>])`: fin de ejecución de una función

12.6.return.clp

```
1 (deffunction signo (?n)
2   (if (> ?n 0)
3   then (return 1)
4   else (if (< ?n 0) then (return -1))))
5 (signo 2)
6 (signo -2)
7 (exit)
```

clips -f 12.6.return.clp

```
1 CLIPS> (deffunction signo (?n)
2   (if (> ?n 0)
3   then (return 1)
4   else (if (< ?n 0) then (return -1))))
5 CLIPS> (signo 2)
6 1
7 CLIPS> (signo -2)
8 -1
```

## ▸ *Otros:*

- ▷ `(progn <exp>*)` evalúa los args y devuelve el valor del último
- ▷ `(progn$ <mcamp-esp> <exp>*)` aplica acciones a cada campo
- ▷ `(break)` rompe la ejecución de un bucle while, loop...
- ▷ `(switch...)` ejecución por casos según valor de exp
- ▷ `(foreach...)` ejecución de acciones para cada campo de un mcampo

# 12.7. Funciones varias

- (random [<startint> <endint>]) y (seed <int>):

12.7.random.clp

```
1 (seed 23)
2 (random 1 6) ; tira dado
3 (random 1 6)
4 (exit)
```

clips -f 12.7.random.clp

```
1 CLIPS> (seed 23)
2 CLIPS> (random 1 6) ; tira dado
3 3
4 CLIPS> (random 1 6)
5 1
```

- (length <cadena-o-mcamp>):
  - length$ hace lo mismo

12.7.length.clp

```
1 (length (create$ a b c d e))
2 (length "gato")
3 (exit)
```

clips -f 12.7.length.clp

```
1 CLIPS> (length (create$ a b c d e))
2 5
3 CLIPS> (length "gato")
4 4
```

- (sort <fcomp> <exp>*)
  - fcomp(?x ?y) TRUE si ordenados

12.7.sort.clp

```
1 (sort > 4 3 5 7 2 7)
2 (deffunction strcmp (?a ?b) (> (str-compare ?a ?b) 0))
3 (sort strcmp Laia Pere Manel Pau)
4 (exit)
```

clips -f 12.7.sort.clp

```
1 CLIPS> (sort > 4 3 5 7 2 7)
2 (2 3 4 5 7 7)
3 CLIPS> (deffunction strcmp (?a ?b) (>= (str-compare ?a ?b) 0))
4 CLIPS> (sort strcmp Laia Pere Manel Pau)
5 (Laia Manel Pau Pere)
```

# 12.9. Funciones para hechos

- `(assert <RHS>+)`: inserta hecho(s); devuelve dirección (FALSE si está)

12.9.assert.clp

```
1 (assert (color rojo))
2 (assert (color verde) (valor (+ 3 4)))
3 (assert (color rojo))
4 (exit)
```

clips -f 12.9.assert.clp

```
1 CLIPS> (assert (color rojo))
2 <Fact-0>
3 CLIPS> (assert (color verde) (valor (+ 3 4)))
4 <Fact-2>
5 CLIPS> (assert (color rojo))
6 FALSE
```

- `(retract <fet>|<int>|*)`: borra hecho(s)

**2.1.hola.clp**

```
1  (deffacts bf (pendiente Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendiente ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6    (retract ?f)
7    (assert (pendiente $?y)))
8  (defrule acaba (pendiente) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

# 13. Órdenes

## 13.1. Órdenes de entorno

- `(load[*] <fichero>)`: carga constructores (* silenciosa)
- `(save <fichero>)`: graba constructores (deffacts y defrules)
- `(clear)`: elimina constructores y datos asociados (agenda)
- `(exit <int>)`
- `(reset)`: reinicia CLIPS (con deffacts y defrules)
- `(batch[*] <fitxer>)`: carga constructores (* silenciosa)
- Otros: `bload bsave options system apropos...`

# 13.2. Órdenes de depuración

- `([un]watch all|globals|rules|activations|facts)`

2.1.hola.clp

```
1  (deffacts bf (pendiente Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendiente ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6    (retract ?f)
7    (assert (pendiente $?y)))
8  (defrule acaba (pendiente) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

# 13.4. Órdenes para hechos

- `(facts)`: muestra la base de hechos (BH)
- `(load-facts <fichero>)`: inserta los hechos del fichero en la BH
- `(save-facts <fichero>)`: graba los hechos de la BH en fichero

# 13.5. Órdenes deffacts

- `(ppdeffacts <nom-deffacts>)`: muestra los hechos indicados
- `(list-deffacts)`: muestra los nombres de todos los `deffacts`
- `(undeffacts <nom-deffacts>)`: borra los hechos indicados

# 13.6. Órdenes defrule

- `(ppdefrule <nom-regla>)`: muestra una regla
- `(list-defrules)`: muestra los nombres de todas las reglas
- `(undefrule <nom-regla)`: borra la regla indicada
- `(matches <nom-regla> [verbose|succint|terse])`
- Otras: `set-break remove-break show-breaks...`

13.6.matches.clp

```
(deffacts bf (f a b c))
(defrule R (f $?x ?y $?z)
  => (printout t "x=" ?x " y=" ?y " z="?z crlf))
(reset)
(matches R)
(reset)
(run)
(exit)

```

clips -f 13.6.matches.clp

```
CLIPS> (deffacts bf (f a b c))
CLIPS> (defrule R (f $?x ?y $?z)
  => (printout t "x=" ?x " y=" ?y " z="?z crlf))
CLIPS> (reset)
CLIPS> (matches R)
Matches for Pattern 1
f-1
f-1
f-1
Activations
f-1
f-1
f-1
CLIPS> (reset)
CLIPS> (run)
x=() y=a z=(b c)
x=(a) y=b z=(c)
x=(a b) y=c z=()
```

# 13.7. Órdenes de agenda

- `(agenda)`: muestra todas las instancias de la agenda
- `(run [<int>])`: ejecuta el número de pasos indicado
- `(halt)`: acaba la ejecución (en la RHS de la regla objetivo)
- `(set-strategy depth|breadth|...)`: resolución conflictos
- `(get-strategy)`: estrategia de resolución de conflictos actual
- `(set-salience-evaluation <val>)`: evalúa prioridades
  - `when-defined`: cuando se definen
  - `when-activated`: cuando se activan
  - `every-cycle`: cada ciclo
- `(get-salience-evaluation)`: criterio de evaluación actual
- `(refresh-agenda)`: re-evalúa prioridades en la agenda

# 13.8. Órdenes defglobal

- `(undefglobal nom-defglobal)`: borra las variables indicadas
- `(show-defglobals)`: muestra los nombres de los `defglobals`
- `(set-reset-globals <bool>)`: `TRUE` por defecto
- `(get-reset-globals)`: `reset` reinicia globales?

# 13.9. Órdenes deffunction

- `(ppdeffunction <nom-deffunction)`: muestra la función
- `(list-deffunctions)`: muestra los nombres de las funciones
- `(undeffunction <nom-deffunction>)`: borra función

# 13.15. Órdenes de análisis computacional

- `(set-profile-percent-threshold [0,100])`: 0 de inicio
- `(get-profile-percent-threshold)`
- `(profile-reset)`: reinicia el análisis computacional
- `(profile-info)`: muestra el análisis
- `(profile constructs | user-functions | off)`

13.15.clp

```
1 (progn (profile user-functions) (run) (profile off)
↪   (profile-info) (exit))
```

clips -f 13.15.clp

```
1 CLIPS> (progn (profile user-functions) (run) (profile off)
  ↪   (profile-info) (exit))
2 Profile elapsed time = 3e-06 seconds
3 Function Name  Entries       Time     %  Time+Kids %+Kids
4 -------------  -------  --------- ------ --------- ------
5 run                  1  0.000001 33.33%  0.000001 33.33%
6 profile              1  0.000001 33.33%  0.000001 33.33%
```

# Referencias


[1] G. Riley. CLIPS: A Tool for Building Expert Systems. URL.

[2] G. Riley. CLIPS: SourceForge Project Page. URL.

[3] C. Culbert et al. CLIPS Reference Manual I: Basic Programming Guide (v6.31). URL.

[4] C. Culbert et al. CLIPS Reference Manual II: Advanced Programming Guide (v6.31). URL.

[5] C. Culbert et al. CLIPS Reference Manual III: Interfaces Guide (v6.31). URL.

[6] J. Giarratano. CLIPS User's Guide (v6.30). URL.