UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Guía de programación en CLIPS

Alfons Juan

DSIC

Departamento de Sistemas Informáticos y Computación

Índice

0 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].

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

1 Introducción [3, sec. 1]

- Guía basada en la ***Guia de programación bàsica en CLIPS*** [3].
- La guía está adaptada para el uso de CLIPS en SIN, con Linux y procesamiento de órdenes por lotes.
- La sección 2 presenta un resumen del lenguaje CLIPS y terminología básica.
- Las secciones 3 a 11 incluyen detalles adicionales sobre el lenguaje CLIPS.
- Los tipos de acciones y funciones que proporciona CLIPS se definen en la sección 12.
- Las órdenes típicamente interactivas se describen en la sección 13.

2 Resumen de CLIPS [3, sec. 2]

2.1 Interacción con CLIPS

Órdenes desde la línea de órdenes:

```
$ clips
         CLIPS (V6.24 06/15/06)
CLIPS> (+ 3 4)                          ; llamada a función
7
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.

Integración con otros lenguajes de programación:

Se describe en [4], pero no lo utilizaremos.

2.1.hola.clp

```
(deffacts bf (pendiente Manel Nora Laia))
(defrule saluda
  ?f <- (pendiente ?x $?y)
  =>
  (printout t "Hola " ?x crlf)
  (retract ?f)
  (assert (pendiente $?y)))
(defrule acaba (pendiente) => (halt))
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

clips -f2 2.1.hola.clp

```
<== f-0     (initial-fact)
==> f-0     (initial-fact)
==> f-1     (pendiente Manel Nora Laia)
==> Activation 0      saluda: f-1
Hola Manel
<== f-1     (pendiente Manel Nora Laia)
==> f-2     (pendiente Nora Laia)
==> Activation 0      saluda: f-2
Hola Nora
<== f-2     (pendiente Nora Laia)
==> f-3     (pendiente Laia)
==> Activation 0      saluda: f-3
Hola Laia
<== f-3     (pendiente Laia)
==> f-4     (pendiente)
==> Activation 0      acaba: f-4
```

clips -f 2.1.hola.clp

```
CLIPS> (deffacts bf (pendiente Manel Nora Laia))
CLIPS> (defrule saluda
  ?f <- (pendiente ?x $?y)
  =>
  (printout t "Hola " ?x crlf)
  (retract ?f)
  (assert (pendiente $?y)))
CLIPS> (defrule acaba (pendiente) => (halt))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (reset)
<== f-0     (initial-fact)
==> f-0     (initial-fact)
==> f-1     (pendiente Manel Nora Laia)
==> Activation 0      saluda: f-1
CLIPS> (run)
Hola Manel
<== f-1     (pendiente Manel Nora Laia)
==> f-2     (pendiente Nora Laia)
==> Activation 0      saluda: f-2
Hola Nora
<== f-2     (pendiente Nora Laia)
==> f-3     (pendiente Laia)
==> Activation 0      saluda: f-3
Hola Laia
<== f-3     (pendiente Laia)
==> f-4     (pendiente)
==> Activation 0      acaba: f-4
```

2.2 Sintaxis del manual de referencia

Se propociona en [3, ap. G] con notación BNF.

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:*** sec. de carácteres imprimibles hasta un ***delimitador***
  `foo  bad_value  127A  456-93-039  @+=-%`
- ***cadena:*** `"foo"  "a and b"  "1 number"`
- ***hecho:*** lista de valores atómicos ref. por posición o nombre;
  dirección: `<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 retorna un valor (***función***) o no (***orden***):

- ***Definidas por el usuario en CLIPS:*** `deffunction`
- ***Predefinidas [3, ap. H]:***
  `!= * ** + - / < <= <> = >= >= abd acos ...`
- ***Genéricas:*** `defgeneric` y `defmethod` (no usamos)
- ***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 en CLIPS
- `defrule:` definición de reglas
- Otros: `defgeneric defmethod defmodule` (no usamos)

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 aceite)`
- ***No ordenados:*** con `deftemplate` (no usamos)
- ***Órdenes:*** `assert retract` (no usamos: `modify duplicate`)
- 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 retornado de `assert` (`modify` y `duplicate`)
- ***Hechos iniciales:*** con `deffacts`

2.4.2 Objetos (no usamos)

2.4.3 Variables globales

Se definen con `defglobal`.

2.5 Representación del conocimiento

2.5.1 Conocimiento heurístico: reglas

Las reglas constan de dos partes, ***el antecedente o parte izquierda (LHS)*** y el ***consecuente o parte derecha (RHS)***:

- La parte izquierda consta de una o más condiciones que han de cumplirse para que se ejecute la parte derecha (acciones).
- Un tipo de condición muy importante es el ***patrón***. Los patrones imponen una serie de restricciones para determinar qué hecho u objetos satisfacen la condición especificada por el patrón.
- El proceso de encaje de hechos u objetos a patrones se llama ***encaje o comprobación de patrones*** (***pattern matching***).
- El ***motor de inferencia*** realiza el encaje de patrones con tal de determinar las reglas que son aplicables.
- Si hay más de una regla aplicable, el motor de inferencia aplica una ***estrategia de resolución de conflictos*** para elegir una.

2.5.2 Conocimiento procedimental

Código como el de los lenguages convencionales:

- ***Funciones definidas por el usuario en CLIPS:*** `deffunction`
- ***Funciones genéricas:*** `defgeneric defmethod` (no usamos)
- ***Otros:*** paso de mensajes a objetos y `defmodules` (no usamos)

2.6 El lenguage orientado a objetos de CLIPS (no usamos)

3 Constructor deftemplate [3, sec. 3] (no usamos)

4 Constructor deffacts [3, sec. 4]

Inserta/reconstruye una lista de hechos cuando se ejecuta `reset`.

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

5 Constructor defrule [3, sec. 5]

Una regla es una colección de condiciones y las acciones a ejecutar si las condiciones se cumplen. Se ejecuta (o dispara, ***fire***) en función de la existencia o no-existencia de ***entidades patrón*** (hechos o instancias de clases definidas por el usuario, ***pattern entities***). El motor de inferencia encaja las reglas al estado actual del sistema (***fact-list*** y ***instance-list***) y aplica las acciones necesarias.

5.1 Definiendo reglas

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

Ejemplo:

5.1.defrule.clp

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

5.2 Ciclo básico de ejecución de reglas

No se tiene un programa convencional (secuencia explícita de operaciones pre-establecida), sino que el ***motor de inferencia*** se encarga de decidir las operaciones a realizar por aplicación del conocimiento (reglas) a los datos (hechos e instancias). Los pasos básicos son:

a) Selección de una regla de la ***agenda***; si no hay, se acaba.

b) Se ejecuta la parte derecha de la regla seleccionada.

c) El paso b) comporta activación y desactivación de reglas: las activadas se añaden a la agenda; las desactivadas se eliminan.

d) Si utilizamos prioridades dinámicas, se re-evaluan las prioridades (`salience`) de las reglas de la agenda y volvemos a a).

En [3] se describen los pasos básicos más detalladamente.

5.3 Estrategias de resolución de conflictos

La agenda incluye todas las reglas con las condiciones satisfechas y no ejecutadas todavía. Se puede ver como una pila: la regla en la cima es la primera que se ejecuta. Cuando se activa una nueva regla, se determina su lugar en la agenda como sigue:

a) Las nuevas reglas se sitúan encima de las de menor prioridad y debajo de las de mayor prioridad.

b) Las reglas de igual prioridad se ordenan de acuerdo con la ***estrategia de resolución de conflictos*** actual.

c) Si una regla se activa juntamente con otras reglas por la misma inserción o borrado de un hecho, y los pasos a) y b) no permiten establecer una ordenación, entonces la regla se ordena arbitrariamente (*no aleatoriamente*) en relación con las otras reglas. Este orden arbitrario puede coincidir con el de definición de las reglas, pero no se aconseja confiar en que así sea.

5.3.1 Profundidad (*depth*)

Las nuevas reglas se sitúan encima de todas las de igual prioridad; es la estrategia por omisión.

5.3.2 Anchura (*breadth*)

Las nuevas reglas se sitúan debajo de todas las de igual prioridad.

5.3.3 Otras estrategias (no usamos)

*Simplicidad (*simplicity*)*, *Complejidad (*complexity*)*, *LEX*, *MEA* i *aleatoria*.

5.4 Sintaxis de la LHS

La LHS de una regla consta de una serie de ***elementos condicionales (CEs)*** que se han de satisfacer para que la regla se añada a la agenda. Hay ocho tipos de CEs:

- ***CEs patrón:*** restricciones sobre los hechos que lo satisfacen
- ***CEs test:*** evalúan expresiones durante el encaje de patrones
- ***CEs and:*** dado un grupo de CEs, todos se han de satisfacer
- ***CEs or:*** dado un grupo de CEs, al menos 1 se ha de satisfacer
- ***CEs not:*** dado un CE, *no* se ha de satisfacer
- ***CEs exists:*** al menos un encaje parcial en un grupo de CEs
- ***CEs forall:*** todos los encajes parciales de un grupo de CEs
- ***CEs logical:*** condiciona la RHS a los encajes en la LHS

```
<conditional-element> ::=
  <pattern-CE> | <assigned-pattern-CE> |
  <not-CE> | <and-CE> | <or-CE> |
  <logical-CE> | <test-CE> |
  <exists-CE> | <forall-CE>
```

5.4.1 CE patrón

El primer campo de un patrón ha de ser un símbolo; para saber si es un hecho ordenado, un no ordenado o un objeto.

Restricciones literales: constantes.

5.4.1.literales.clp

```
(deffacts data-facts
(data 1.0 blue "red")
(data 1 blue)
(data 1 blue red)
(data 1 blue RED)
(data 1 blue red 6.9))
(defrule find-data (data 1 blue red) =>)
(watch facts)
(watch activations)
(reset)
(exit)
```

clips -f2 5.4.1.literales.clp

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

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

5.4.1.comodines.clp

```
(deffacts data-facts
(data 1.0 blue "red")
(data 1 blue)
(data 1 blue red)
(data 1 blue RED)
(data 1 blue red 6.9))
(defrule find-data (data ? blue red $?) =>)
(watch facts)
(watch activations)
(reset)
(exit)
```

clips -f2 5.4.1.comodines.clp

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

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

5.4.1.variables.clp

```
(deffacts data-facts (data 1 blue) (data 1 blue red)
  (data 1 blue red 6.9))
(defrule find-data-1
  (data ?x $?y ?z)
  => (printout t "?x=" ?x " $?y=" $?y " ?z=" ?z crlf ))
(watch facts)
(watch activations)
(set-strategy breadth) ; por omisión es depth
(reset)
(run)
(exit)
```

clips -f2 5.4.1.variables.clp

```
==> f-0     (initial-fact)
==> f-1     (data 1 blue)
==> Activation 0      find-data-1: f-1
==> f-2     (data 1 blue red)
==> Activation 0      find-data-1: f-2
==> f-3     (data 1 blue red 6.9)
==> Activation 0      find-data-1: f-3
?x=1 $?y=() ?z=blue
?x=1 $?y=(blue) ?z=red
?x=1 $?y=(blue red) ?z=6.9
```

Restricciones conectivas: `&`(y), `|`(o), `~`(no) (no usamos)

Restricciones de predicado: `:<func>` (no usamos)

Restricciones de valor de retorno: `=<func>`

5.4.1.retorno.clp

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

clips -f2 5.4.1.retorno.clp

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

Encaje de patrones con patrones objeto: (no usamos)

Direcciones de patrón: ?<var> <- <pattern-CE>

5.4.1.direcciones.clp

```
(deffacts bf (color rojo) (color verde))
(defrule encuentra-color
  ?f <- (color ?c)
  => (printout t ?c " encontrado en el hecho " ?f crlf))
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

clips -f2 5.4.1.direcciones.clp

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

5.4.2 CE test

El CE `(test <func>)` se satisface sí `<func>` no devuelve falso.

5.4.2.test.clp

```
(deffacts bf (tenemos 6 platos) (tenemos 5 vasos))
(defrule tenemos-mas
  (tenemos ?n ?x)
  (tenemos ?m ?y)
  (test (> ?n ?m))
  =>
  (printout t "Tenemos más " ?x " que " ?y crlf))
(watch facts)
(watch activations)
(reset)
(run)
(exit)
```

clips -f2 5.4.2.test.clp

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

5.4.3 CE or

El CE `(or <CE>+)` se satisface si cualquiera de los `<CE>+` lo hace.

5.4.3.or.clp

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

clips -f2 5.4.3.or.clp

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

5.4.4 CE and

El CE `(and <CE>+)` se satisface si todos los `<CE>+` lo hacen.

5.4.4.and.clp

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

clips -f2 5.4.4.and.clp

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

5.4.5 CE not

El CE `(not <CE>+)` se satisface sí `<CE>` no lo hace.

5.4.5.not.clp

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

clips -f2 5.4.5.not.clp

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

5.4.6 CE exists (no usamos)

5.4.7 CE forall (no usamos)

5.4.8 CE logical (no usamos)

5.4.9 Reemplazo automático de CEs (ignorad)

5.4.10 Declaración de propiedades de regla

```
<declaration>     ::= (declaro <rule-property>+)
<rule-property>   ::= (salience <integer-expression>) |
                      (auto-foco <boolean-symbol>)
<boolean-symbol> ::= TRUE | FALSE
```

5.4.10.salience.clp

```
(deffacts bf (hecho-estado-obj a b))
(defrule obj
  (declare (salience 1)) ; entre -10000 y +10000; 0 por omisión
  (hecho-estado-obj a b)
  =>
  (printout t "Solución encontrada!" crlf))
(reset)
(run)
(exit)
```

6 Constructor defglobal [3, sec. 6]

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

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

Las utilizaremos para pocos propósitos como por ejemplo contar nodos insertados o limitar la profundidad del árbol de búsqueda.

6.defglobal.clp

```
(defglobal ?*N* = 0)
(deffacts bf (L a b a b a))
(defrule R
  ?f <-(L ?x $?y ?x $?z)
  =>
  (printout t ?x" "?y" "?z crlf)
  (retract ?f)
  (assert (L $?y ?x $?z))
  (bind ?*N* (+ ?*N* 1)))
(watch facts)
(watch activations)
(watch globals)
(set-strategy breadth)
(reset)
(run)
(printout t "N=" ?*N* crlf)
(exit)
```

clips -f2 6.defglobal.clp

```
:== ?*N* ==> 0 <== 0
==> f-0     (initial-fact)
==> f-1     (L a b a b a)
==> Activation 0      R: f-1
==> Activation 0      R: f-1
a (b a b) ()
<== f-1     (L a b a b a)
<== Activation 0      R: f-1
==> f-2     (L b a b a)
==> Activation 0      R: f-2
:== ?*N* ==> 1 <== 0
b (a) (a)
<== f-2     (L b a b a)
==> f-3     (L a b a)
==> Activation 0      R: f-3
:== ?*N* ==> 2 <== 1
a (b) ()
<== f-3     (L a b a)
==> f-4     (L b a)
:== ?*N* ==> 3 <== 2
N=3
```

7 Constructor deffunction [3, sec. 7]

`deffunction` define funciones de usuario.

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

Hemos de pasarle tantos argumentos como paràmetros convencionales (variables mono-evaluadas) tenga y, si tiene un último paràmetro comodín (variable multi-evaluada), podemos pasarle tantos argumentos adicionales como queramos.

7.deffunction.clp

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

clips -f2 7.deffunction.clp

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

8 Funciones genéricas [3, sec. 8] (no usamos)

Definidas con `defgeneric` y `defmethod`.

9 El lenguaje OO de CLIPS [3, sec. 9] (ignorad)

Se detalla en [3, sec. 9].

10 Constructor defmodule [3, sec. 10] (ignorad)

`defmodule` define ***módulos CLIPS*** [3, sec. 10].

11 Atributos de restricción [3, sec. 11] (no usamos)

Restringen los valores que pueden tomar los campos de hechos ordenados y objetos. [3, sec. 11].

12 Acciones y funciones [3, sec. 12]

12.1 Funciones predicado

12.1.pruebas.clp

```
(numberp 23)                    ; prueba de número (entero o real)
(floatp 3.0)                    ; prueba de real
(integerp 3)                    ; prueba de entero
(lexemep SIN)                   ; prueba de cadena o símbolo
(stringp "SIN")                 ; prueba de cadena
(symbolp SIN)                   ; prueba de símbolo
(evenp 2)                       ; prueba de número par
(oddp 3)                        ; prueba de número impar
(multifieldp (create$ a b)) ; prueba de multicampo
;(pointerp <expression>) prueba de dirección externa (ignorad)
(exit)
```

clips -f 12.1.pruebas.clp

```
CLIPS> (numberp 23)                    ; prueba de número (entero o real)
TRUE
CLIPS> (floatp 3.0)                    ; prueba de real
TRUE
CLIPS> (integerp 3)                    ; prueba de entero
TRUE
CLIPS> (lexemep SIN)                   ; prueba de cadena o símbolo
TRUE
CLIPS> (stringp "SIN")                 ; prueba de cadena
TRUE
CLIPS> (symbolp SIN)                   ; prueba de símbolo
TRUE
CLIPS> (evenp 2)                       ; prueba de número par
TRUE
CLIPS> (oddp 3)                        ; prueba de número impar
TRUE
CLIPS> (multifieldp (create$ a b)) ; prueba de multicampo
TRUE
CLIPS> ;(pointerp <expression>) prueba de dirección externa (ignorad)
```

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 número distinto al resto
TRUE
CLIPS> (> 5 4 3)               ; TRUE si args en orden decreciente
TRUE
CLIPS> (>= 5 5 3)              ; TRUE si args en orden no creciente
TRUE
CLIPS> (< 3 4 5)               ; TRUE si args en orden creciente
TRUE
CLIPS> (<= 3 5 5)              ; TRUE si args en orden no decreciente
TRUE
```

clips -f 12.1.logicas.clp

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

12.2 Funciones multicampo

clips -f 12.2.clp

```
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

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 ...

clips -f 12.4.clp

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

12.5 Funciones matemáticas

clips -f 12.5.clp

```
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:

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+
(deffunction f() (bind ?a 3) (bind ?a (- ?a 1)) ?a)
CLIPS> (f)
2
```

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

12.6.ifthenelse.clp

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

```
CLIPS> (load "12.6.ifthenelse.clp")
Defining defglobal: prof
Defining deffunction: inicio
TRUE
CLIPS> (inicio)
Profundidad maxima: 50
Anchura (1) o Profundidad (2): 1
depth
CLIPS> (exit)
```

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

clips -f 12.6.while.clp

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

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

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

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

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

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

clips -f 12.6.return.clp

```
CLIPS> (deffunction signo (?n)
  (if (> ?n 0)
  then (return 1)
  else (if (< ?n 0) then (return -1))))
CLIPS> (signo 2)
1
CLIPS> (signo -2)
-1
```

Otros:

`(progn <exp>*)`: evalúa los args y vuelve el valor del último

`(progn$ <mcampo-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 por cada campo de un mcampo

12.7 Funciones varias

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

clips -f 12.7.random.clp

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

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

clips -f 12.7.length.clp

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

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

clips -f 12.7.sort.clp

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

Otros:

`(gensym)` genera símbolos de la forma `gen<int>` (no usamos)

`(gensym*)` similar a `(gensym)` (no usamos)

`(setgen <int>)` define contador inicial de `(gensym[*])`

`(time)`: segundos transcurridos desde el inicio del sistema

`(get-function-restrictions <func>)`: (no usamos)

`(funcall <func> <exp>)`: construye-evalúa un llamamiento a func

`(timer <exp>*)`: segundos para evaluar las expresiones

`(operating-system)`: sistema operativo

`(local-time)`: hora local

`(gm-time)`: hora GMT

12.8 Funciones deftemplate (no usamos)

12.9 Funciones para hechos

`(assert <RHS>+)`: inserta hecho(s); vuelve dir. (FALSE si está)

clips -f 12.9.assert.clp

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

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

regla saluda de hola.clp

```
(defrule saluda
  ?f <- (pendiente ?x $?y)
  =>
  (printout t "Hola " ?x crlf)
  (retract ?f)
  (assert (pendiente $?y)))
```

Otros: `modify duplicate assert-string fact-index fact-existp fact-relation` ...

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

12.10 Funciones deffacts (no usamos)

12.11 Funciones defrule

`(get-defrule-list)`: mcampo con nombres de las reglas

12.12 Funciones de agenda (no usamos)

12.13 Funciones defglobal

`(get-defglobal-list)`: mcampo con las vars globales

12.14 Funciones deffunction

`(get-deffunction-list)`: mcampo funciones de usuario

12.15 Funciones para funciones genéricas (no usamos)

12.16 Funciones COOL (no usamos)

12.17 Funciones defmodule (no usamos)

12.18 Expansión de secuencias (ignorad)

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[*] <fichero>)`: carga constructores (* silenciosa)

Otros: `bload bsave options system apropos ...`

13.2 Órdenes de depuración

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

13.3 Órdenes deftemplate (no usamos)

13.4 Órdenes para hechos

`(facts)`: muestra la base de hechos (BF)
`(load-facts <fichero>)`: inserta hechos del fichero en BF
`(save-facts <fichero>)`: graba hechos de BF en fichero
`set-fact-duplication get-fact-duplication ppfact`

13.5 Órdenes deffacts

`(ppdeffacts <nom-deffacts>)`: muestra 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])`

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=()
```

Otros: `set-break remove-break show-breaks ...`

13.7 Órdenes de agenda

`(agenda)`: muestra todas las instancias de la agenda

`(run [<int>])`: ejecuta el nombe 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>)`:
establece criterio de evaluación de prioridades
`when-defined`, `when-activated` o `every-cycle`

`(get-salience-evaluation)`: criterio de evaluación actual

`(refresh-agenda)`: re-evalúa prioridades en la agenda

Otros: `focus` `list-focus-stack` `clear-focus-stack`

13.8 Órdenes defglobal

`(undefglobal nom-defglobal)`: borra las vbles indicadas

`(show-defglobals)`: muestra nombres de todos los `defglobals`

`(set-reset-globals <bool>)`: `TRUE` por omisión

`(get-reset-globals)`: `reset` reinicia globales?

13.9 Órdenes deffunction

`(ppdeffunction <nom-deffunction)`: muestra la función

`(list-deffunctions)`: muestra nombres de las funciones

`(undeffunction <nom-deffunction>)`: borra función

13.10 Órdenes para funciones genéricas (no usamos)

13.11 Órdenes COOL (no usamos)

13.12 Órdenes defmodule (no usamos)

13.13 Órdenes de gestión de memoria (no usamos)

13.14 Manipulación de texto externo (no usamos)

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)`

```
(progn (profile user-functions) (run) (profile off) (profile-info))
```

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.