CLIPS

Introducción a CLIPS (II)

Contenido

1. Hechos (no ordenados).
2. Variables.
3. Ampliando el lenguaje de equiparación.
4. Plantilla objeto-atributo-valor.
5. Ejercicios.

1. Hechos

Hechos ordenados

- Hechos ordenados:
  - (status walking)
  - (persona nombre "Luis Prieto" edad 53 altura 1.83)
  - (persona "Luis Prieto" 53 1.83)

- No hay que definir su estructura
  - Hechos iniciales mediante constructor **deffacts**
  - Añadir dinámicamente mediante función **assert**

Hechos (no ordenados)

- Constan de nombre-relación, seguido de ninguno o más slots y su valores asociados

```
(persona      (nombre Luis Perez Rubio)
              (Edad 23)
              (Color-ojos azul)
              (Color-pelo rubio))
```

- Hay que definir su estructura mediante el constructor **deftemplate**

Constructor Deftemplate

- Define la estructura del hecho

(deftemplate <nombre-relación> [<comentario>]
    <Definición-slot>*)

    <Definición-slot>* ::= (slot <nombre-slot>) |
                           (multislot <nombre-slot>)

Ejemplo plantilla hecho

```
(deftemplate persona
    "Un ejemplo de plantilla"
    (multislot nombre)
    (slot edad)
    (slot color-ojos)
    (slot color-pelo))
```

Definir hechos en Base de Conocimiento

```
(deffacts colegas "Algunos colegas"
    (persona       (nombre Luis Perez Rubio)
                   (edad 23)
                   (color-ojos azul)
                   (color-pelo rubio))
    (persona       (nombre Juan Soto Garcia)
                   (edad 24)
                   (color-ojos gris)
                   (color-pelo castaño))
)
```

Confrontación con hechos

- Basta con que equiparen los slots del hecho que referencia el patrón en el antecedente de las reglas

```
(defrule edad-colegas "Encontrar la edad de los colegas"
    (persona (nombre ?x ? ?)
                    (edad ?y))
=>
    (printout t " " ?x " tiene " ?y " años." crlf)
)
```

```
;  Ejemplo hecho: Colegas

;  Definir plantilla

(deftemplate persona
      "Un ejemplo de plantilla"
      (multislot nombre)
      (slot edad)
      (slot color-ojos)
      (slot color-pelo ))

;     Hechos iniciales

(deffacts colegas "Algunos colegas"
      (persona            (nombre Luis Perez Rubio)
                          (edad 23)
                          (color-ojos azul)
                          (color-pelo rubio))
      (persona            (nombre Juan Soto Garcia) (edad 24)(color-ojos gris) (color-pelo castaño))
)

; Única regla

(defrule edad-colegas "Encontrar la edad de los colegas"
      (persona (nombre ?x ? ?)
                          (edad ?y))
=>
      (printout t " " ?x " tiene " ?y " años." crlf)
)
```

Ejecución

```
(load "colegas.clp")
(reset)
(run)
 Juan tiene 24 años.
 Luis tiene 23 años.
```

Manipulación de hechos

- Las operaciones sobre hechos, salvo **assert**, requieren la obtención de un índice al hecho: **fact-address**

- Una variable se puede ligar al **fact-address** del hecho que confronta con el patrón de una regla, mediante el operador **pattern binding**:<-

  ?x <- (persona (edad ?y))

- ?x contiene una cadena de caracteres del estilo a <Fact-1>, es decir, <Fact-numero>

- Para extraer solo el índice del hecho usar **fact-index**:

  (fact-index ?x)

```
(deftemplate persona    (slot nombre)   (slot direccion))
(deftemplate mudanza    (slot nombre)   (slot direccion))


(defrule procesar-informacion-mudanza
   ?x <- (mudanza (nombre ?nombre) (direccion ?direccion))
   ?y <- (persona (nombre ?nombre))
   =>
   (retract ?x)
   (modify ?y (direccion ?direccion))
)


(deffacts informacion-mudanza
   (persona (nombre "Juan Rodrigo") (direccion "Paseo Zorrilla 37"))
   (mudanza (nombre "Juan Rodrigo") (direccion "Calle Gaviota 12"))
)
```

Ejecución

```
(clear)
(load "mudanza.clp")
(reset)
(facts)
f-0     (initial-fact)
f-1     (persona (nombre "Juan Rodrigo") (direccion "Paseo Zorrilla 37"))
f-2     (mudanza (nombre "Juan Rodrigo") (direccion "Calle Gaviota 12"))
For a total of 3 facts.
(run)
(facts)
f-0     (initial-fact)
f-3     (persona (nombre "Juan Rodrigo") (direccion "Calle Gaviota 12"))
For a total of 2 facts.
```

2. Variables

Variables Multicampo

- Su ligadura es una secuencia de valores
- Sintaxis lado izquierdo de una regla: $?< nombre-variable >
- Sintaxis lado derecho de una regla: se puede suprimir $

```
(defrule print-hijos
   (print-hijos $?nombre)
   (persona (nombre $?nombre)
   (hijos $?hijos))
   =>
   (printout t ?hijos " son hijos de " $?nombre "." crlf))
```

```
(deftemplate persona   (multislot nombre)   (multislot hijos))

(deffacts algunas-personas
    (persona   (nombre Juan Rodrigo) (hijos Pablo Ana))
    (persona  (nombre Jose Gimenez)  (hijos Pedro)))

(defrule print-hijos
    (print-hijos $?nombre)
    (persona (nombre $?nombre)
    (hijos $?hijos))
   =>
   (printout t ?hijos " son hijos de " $?nombre "." crlf))
```

NOTA: Ver si funciona así este programa, y si no, ver qué se puede hacer.

Ejecución

(clear)
(load “VariablesMulticampo.clp")
(reset)
(assert (print-hijos Juan Rodrigo))
<Fact-3>
(run)
(Pablo Ana) son hijos de (Juan Rodrigo).

Variables globales

<variable-global> ::= ?*<símbolo>*

- Es preciso declararlas

(defglobal ?*contador* = 0)

- Se pueden utilizar en los patrones de las reglas.
- Su valor se modifica con la función **bind**, pero no genera nuevas confrontaciones.

(bind ?*contador* 3)

Ligadura de variables locales

```
(defrule solicitar-nombre
    (declare (salience 10000))
=>
    (printout t  "Nombre: ")
    (bind ?respuesta (read))
    (assert (nombre-usuario ?respuesta))
)
```

3. Ampliando el lenguaje de equiparación

Restricciones

- Internas a los patrones
- Las restricciones, **field constraint**, permiten restringir los valores de un campo que satisfacen la confrontación de patrones.
  - Not: ~
  - Or: |
  - And: &

Restricciones not *(~)*, or *(|)*

```
(defrule persona-sin-pelo-moreno
   (persona (nombre ?nombre) (pelo ~moreno))
   =>
   (printout t ?nombre " no tiene el pelo moreno." crlf))

(defrule persona-con-pelo-moreno-o-rubio
   (persona (nombre ?nombre) (pelo negro | rubio))
   =>
   (printout t ?nombre " tiene el pelo moreno o rubio." crlf))
```

Restricción and (&)

- Habitualmente, junto a otras restricciones

```
(defrule pelo-moreno-o-rubio-y-otro-color
    (persona (nombre ?nombre1)
                (pelo ?color1 & moreno | rubio))
    (persona (nombre ?nombre2)
                 (pelo ?color2 & ~?color1))
   =>
   (printout t ?nombre1 " tiene el pelo " ?color1 "." crlf)
   (printout t ?nombre2 " tiene el pelo " ?color2 "." crlf)
)
```

Restricción predicado

- Permiten evaluar predicados en los patrones.
- Junto a la restricción and (&)

```
(defrule mayor-de-edad
   (persona    (nombre ?nombre)
               (edad ?edad & :(>= ?edad 18)))
=>
   (printout t ?nombre " es mayor de edad." crlf)
)
```

Parte izquierda de la regla: or

- Condición **or**

```
(defrule cortar-electricidad
    (or  (emergencia (tipo inundacion))
         (emergencia (tipo fuego)))
=>
    (printout t "Desconectar electricidad." crlf)
)
```

- Cuidado: se comporta como dos reglas.

Parte izquierda de la regla: and

- Condición **and**
  - Implícita. Las siguientes reglas son equivalentes:

```
(defrule cortar-electricidad-una-vez
    (and (potencia-electrica (estado on))
         (emergencia (tipo fuego)))
=>
    (printout t "Desconectar electricidad." crlf))
```

```
(defrule cortar-electricidad-una-vez
    (potencia-electrica (estado on))
    (emergencia (tipo fuego))
=>
    (printout t "Desconectar electricidad." crlf))
```

Parte izquierda de la regla: test

- Condición **test**
- Permite evaluar predicados en el antecedente de las reglas

```
(defrule comprobar-edad-par
   (persona (nombre ?nombre)
            (edad ?edad))
   (test (and (numberp ?edad)
              (eq 0 (mod ?edad 2))))
=>
   (printout t "La edad de " ?nombre " es par." crlf)
)
```

Parte izquierda de la regla: not (I)

- Condición **not**
- (not <patron>)
  - Se satisface si patrón no confronta
  - Puede contener variables
    - Ligadas: han tomado su valor antes del not
    - No ligadas: toman su valor por primera vez en el not.
      - En este caso el alcance de las posibles ligaduras es local al *not*

```
(defrule sin-emergencia
    (informar)
    (not (emergencia))
=>
    (printout t "No hay ninguna emergencia." crlf))
```

Parte izquierda de la regla: not (II)

- Condición **not**

```
(deftemplate emergencia
        (slot tipo))

(deffacts informar (informar-estado))

(defrule informar-emergencia
      (informar-estado)
      (emergencia (tipo ?tipo))
      =>
      (printout t "Manejando " ?tipo " emergencia" crlf))

(defrule no-hay-emergencia
      (informar-estado)
      (not (emergencia))
      =>
      (printout t "No se estan manejando emergencias" crlf))
```

Parte izquierda de la regla: not (III)

- Condición **not**

```
(deftemplate emergencia
        (slot tipo))

(deffacts informar (informar-estado))

(defrule informar-emergencia
     (informar-estado)
      (emergencia (tipo ?tipo))
      =>
      (printout t "Manejando " ?tipo " emergencia" crlf))

(defrule no-hay-emergencia
     (informar-estado)
      (not (emergencia (tipo ?tipo)))
      =>
      (printout t "No hay emergencia de tipo " ?tipo  crlf))
```

La variable ?tipo es local al not ya que es el primer lugar donde toma valor, y no se conserva su valor en la parte derecha de la regla (RHS). Por tanto, aparece un error al cargar el programa en CLIPS.

Parte izquierda de la regla: not (IV)

- Condición **not**

```
(defrule numero-mayor
    (numero ?x)
    (not (numero ?y & :(> ?y ?x)))
=>
    (printout t "El número mayor es: " ?x crlf)
)
```

En este caso la variable ?x toma valor antes del not, por tanto, conserva su valor en la parte derecha de la regla.

4. Plantilla objeto-atributo-valor

Plantilla objeto-atributo-valor

- La conceptualización objeto-atributo-valor se puede representar en CLIPS con las siguientes plantillas:
  - Univaluado

```
(deftemplate   oav-u
   (slot objeto (type SYMBOL))
   (slot atributo (type SYMBOL))
   (slot valor))
```

  - Multivaluado

```
(deftemplate   oav-m
   (slot objeto (type SYMBOL))
   (slot atributo (type SYMBOL))
   (slot valor))
```

Plantilla objeto-atributo-valor
Atributos multivaluados

- Basta con definir una plantilla para ellos
- Multivaluado:

```
(deftemplate   oav-m
   (slot objeto (type SYMBOL))
   (slot atributo (type SYMBOL))
   (slot valor))
```

Semántica multivaluada

```
(deffacts hechos-multivaluados
      (oav-m (objeto Juan)
                  (atributo sintoma)
                  (valor fiebre))
      (oav-m (objeto Juan)
                  (atributo sintoma)
                  (valor dolorCabeza))
)
```

- ¿Síntomas de juan?

Plantilla objeto-atributo-valor
Atributos univaluados

- Necesitamos una plantilla
- Univaluado:

```
(deftemplate   oav-u
   (slot objeto (type SYMBOL))
   (slot atributo(type SYMBOL))
   (slot valor))
```

- **Y garantizar la semántica univaluada**

Semántica ¿univaluada?

```
(deffacts hechos-no-univaluados
      (oav-u (objeto Juan)
                  (atributo edad)
                  (valor 35))
      (oav-u (objeto Juan)
                  (atributo edad)
                  (valor 41))
)
```

- ¿Edad de Juan?

5. Ejercicios.

Ejercicios

1. Proponer una solución computacional en CLIPS para garantizar la semántica univaluada de los tripletes Objeto-Atributo-Valor univaluados. Codificar la solución en CLIPS y probarla con un ejemplo sencillo.

   Sugerencia: garantizar que en todo momento en la memoria de trabajo solo se mantiene el último hecho añadido

2. Ver el enunciado en el archivo SP-Cardio.