Inteligencia en Redes de Comunicaciones

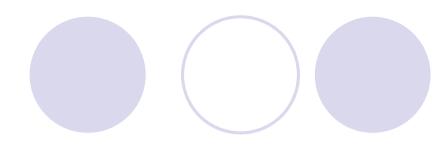
CLIPS

Julio Villena Román

jvillena@it.uc3m.es



Índice



- Lenguaje CLIPS:
 - Hechos, reglas, variables, funciones, operadores, referencias...
 - Estrategias de resolución de conflictos
 - Clases y objetos
 - Plantillas
 - Módulos
- Jess



CLIPS (C Language Integrated Production System)

- Herramienta para el desarrollo de sistemas expertos desarrollada por la NASA desde 1986
- Sistema de producción que incluye:
 - un sistema de mantenimiento de verdad con encadenamiento hacia delante
 - adición dinámica de reglas y hechos
 - diferentes estrategias de resolución de conflictos
- Es fácilmente integrable en aplicaciones en diferentes lenguajes y está disponible en diversas plataformas
- Incluye COOL (Clips Object-Oriented Language)
- Extensiones para Lógica Borrosa (FuzzyCLIPS)



Hechos (facts)

- Patrones que pueden tener un campo o varios
- Los componentes de un patrón pueden ser de diferentes tipos: numéricos, cadenas, símbolos, ...

```
(nombre "Juan") (edad 14)
```

 Si tienen varios campos, el primero suele representar una relación entre los restantes

```
(alumnos Juan Luis Pedro)
```

- Se pueden añadir dinámicamente a la memoria de trabajo con assert y quitarlos con retract
- Cada hecho tiene asignado un identificador único en la fact-list



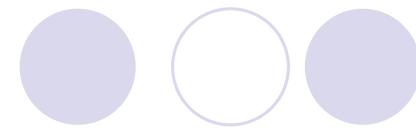
Agrupación de hechos

- La estructura deffacts se utiliza para agrupar conceptualmente diferentes hechos sobre el mismo objeto
- Ejemplo:

```
(deffacts hechos-del-vehiculo
    "información del vehículo"
    (marca Citroen)
    (modelo Xsara Picasso)
    (puertas 5)
)
```



Reglas



```
(defrule <nombre-regla>
  [<descripción opcional>]
  [(declare (salience <num>))]
  (patrón 1)
  (patrón 2)
  (patrón N)
=>
  (acción 1)
  (acción 2)
  (acción N)
```



Ejemplo (hechos y reglas)

```
;; * EJEMPLO DE HECHOS Y REGLAS *
(deffacts nombres
 (nombre Pedro Perez Perez)
 (nombre Luis Montes Garcia)
 (nombre Carlos Marin Rodriguez)
 (nombre Juan Soler Leal)
(defrule apellidos-iguales
 (nombre ?x ?y ?y)
=>
 (printout t
 "D." ?x " tiene los dos apellidos
  iquales" crlf)
); no poner retorno en cadenas
```

```
(defrule buenos-dias
  (nombre ? ?x ?)
=>
  (printout t
  "Buenos días, Sr." ?x crlf)
(defrule buenas-tardes
  (nombre $? ?x ?)
=>
  (printout t
  "Buenas tardes, Sr." ?x crlf)
```



Motor de inferencias

- El motor de inferencias trata de emparejar la lista de hechos con los patrones de las reglas
- Si todos los patrones de una regla están emparejados se dice que dicha regla está activada
- La agenda almacena la lista de activaciones, por orden de prioridad
- Para insertar una activación en la agenda, se siguen las estrategias de resolución de conflictos



Órdenes básicas del motor de inferencias

- (facts): lista los hechos de la MT
- (assert <hecho>): añade el hecho a la MT
- (retract <ref-hecho>): elimina el hecho de la MT
- (clear): elimina todos los hechos de la MT
- (reset): elimina todos los hechos de la MT, las activaciones de la agenda y restaura las condiciones iniciales:
 - añade el hecho initial-fact y el objeto initial-object
 - añade los hechos y ejemplares iniciales definidos con deffacts y definstances
 - añade las variables globales definidas con defglobal
 - fija como módulo actual el módulo MAIN

Ejecución de un programa...

- Editar las reglas/hechos con un editor de textos
- Cargarlo con load: (load "ej.clp")
- Ejecutarlo (normalmente después de un reset previo): run
- Si hay modificaciones, la base de conocimientos se puede almacenar con save: (save "ej.clp")



Resolución de conflictos

- Cuando una regla es activada, se coloca en la agenda según los siguientes criterios:
 - Las reglas más recientemente activadas se colocan encima de las reglas con menor prioridad, y debajo de las de mayor prioridad
 - 2. Entre reglas de la misma prioridad, se emplea la estrategia configurada de resolución de conflictos
 - 3. Si varias reglas son activadas por la aserción de los mismos hechos, y no se puede determinar su orden en la agenda según los criterios anteriores, se insertan de forma arbitraria (no aleatoria)



Estrategias de resolución de conflictos

- Supongamos:
 - hecho-a activa r1 y r2
 - hecho-b activa r3 y r4
 - añadimos a MT hecho-a y hecho-b en este orden
- Estrategia en profundidad (Depth)
 - Es la estrategia por defecto
 - O Agenda → r3, r4, r1, r2
- Estrategia en anchura (Breadth)
 - O Agenda → r1, r2, r3, r4



Estrategias de resolución de conflictos (2)

- Estrategia de simplicidad/complejidad
 - El criterio de ordenación es la especificidad de la regla, esto es, el número de comparaciones que deben realizarse en el antecedente

- Estrategia aleatoria
 - A cada activación se le asigna un número aleatorio para determinar su orden en la agenda.
 - Sin embargo, siempre se le asigna el mismo número en diferentes ejecuciones

Estrategias de resolución de conflictos (3)

Estrategia LEX

- Se asocia a cada hecho y ejemplar el tiempo en que fueron creados
- Se da mayor prioridad a las reglas con un hecho más reciente, comparando los patrones en orden descendente

Estrategia MEA

 Se aplica la misma estrategia de LEX, pero mirando sólo el primer patrón

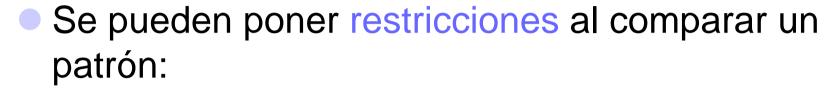


Variables

- Es posible utilizar variables en los patrones de la parte izquierda de una regla
- Las variables comienzan por ?



Patrones avanzados



```
negación (~) (color ~rojo)
```

- Conjunción (&) (color rojo&amarillo)
- disyunción (|) (color rojo|amarillo)
- También se pueden unir patrones con las relaciones lógicas or, and y not
 - por defecto, los patrones se unen con and



Ejemplo (patrones avanzados)

```
(defrule regla-tonta
(defrule no-cruzar
                                           (luz verde&~roja)
  (luz ~verde)
                                         =>
=>
                                           (printout t "Luz verde" crlf)
  (printout t "No cruce" crlf)
                                         (defrule precaucion
                                           (luz ?color&amarillo|intermitente)
(defrule precaucion
                                         =>
  (luz amarilla | intermitente)
                                           (printout t "Cuidado luz " ?color
=>
                                            crlf)
  (printout t "Cruce con precaución"
   crlf)
                                         (defrule no-cruzar
                                           (estado caminando)
                                           (or (luz roja)
(defrule regla-imposible
                                           (policia dice no cruzar)
  (luz verde&roja)
                                           (not (luz ?))); sin luz
                                         =>(printout t "No cruzar" crlf)
=>
  (printout t ";;MILAGRO!!" crlf)
```



Evaluación de patrones

- Hay dos funciones de evaluación:
 - (test <función-booleana> <arg>)
 - ?variable&:(<función-booleana> <arg>)
- Funciones booleanas (predefinidas/usuario):
 - O lógicas: or, not, and
 - comparación numérica: =,<>, >=, >, <=, <</p>
 - comparación de otro tipo: eq, neq
 - funciones predicado: stringp, numberp, evenp, lexemep, symbolp, ...



Ejemplo (evaluación de patrones)

```
(defrule mes-valido-1
  (entrada ?numero)
  (test (and (>= ?numero 1) (<= ?numero 12)))
=>
    (printout t "Mes válido" crlf)
)
(defrule mes-valido-2
  (entrada ?numero&:(and (>= ?numero 1) (<= ?numero 12)))
=>
    (printout t "Mes válido" crlf)
)
```



Variables globales (defglobal)

 Permiten almacenar valores accesibles en reglas y funciones, muy útiles para resultados (defglobal

```
?*num* = 3
?*suma* = (+ ?*num* 2)
?*cadena* = "hola"
?*lista* = (create$ a b c)
```

 Pueden aparecer en la parte izquierda de las reglas si no son utilizadas para asignar un valor y su cambio no activa las reglas, pero no pueden ser parámetros de funciones ni métodos



Referencias

 Con el operador <- se almacena una referencia a un hecho en una variable

```
(defrule matrimonio
    ?soltero <- (soltero ?nombre)
=>
    (printout t ?nombre "está solter@" crlf)
    (retract ?soltero)
    (assert (casado ?nombre))
    (printout t ?nombre " ahora está casad@" crlf)
```



Asignación de valores a variables

Sintaxis:

```
(bind <variable> <valor>)

(defrule suma
    (numeros ?x ?y)
=>
    (bind ?r (+ ?x ?y))
    (assert (suma-es ?r))
    (printout t ?x " + " ?y " = " ?r crlf)
)
```



Funciones

```
(deffunction <nombre-fun>
  [comentario]
  (?arg1 ?arg2 ...?argM)
  (<acción 1>
  ...
  <acción K>)
```

- Devuelven el resultado de la última acción
- printout: permite mostrar un mensaje por un dispositivo (t = salida estándar)
- read (lee una palabra) y readline (lee una línea)



Ejemplo (funciones)



Estructuras de control



```
(if (<condición>)
  then (<acciones>)
  [else (<acciones>)]
)
```

Bucle

```
(while (<condición>)
        (<acciones>)
)
```



Ejemplo (estructuras de control)

(defrule continua-bucle



Clases y objetos

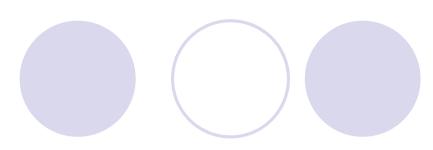
- La construcción defclass especifica las ranuras (atributos) de una nueva clase de objetos
- Las facetas especifican propiedades de una ranura, por ejemplo, el tipo de valor
- La construcción defmessage-handler crea los elementos procedimentales (métodos) de una clase de objetos
- La herencia múltiple se utiliza para especializar una clase existente (la nueva clase hereda las ranuras y los métodos de sus superclases)



Ejemplo (clases)

(defclass perro

```
(is-a USER)
(role concrete)
(pattern-match reactive)
(slot patas (type INTEGER)
  (default 4)(range 0 4)
  (create-accessor read-write))
(slot raza (type SYMBOL)
  (allowed-symbols caniche dogo)
  (default caniche)
  (create-accessor read-write))
(multislot dueno (type STRING)
  (cardinality 1 2)
  (default "Juan")
  (create-accessor read-write))
(slot ident(type SYMBOL)
  (default-dynamic (gensym)))
```



```
CLIPS> (make-instance Lulu of perro)
[Lulu]
CLIPS> (send [Lulu] print)
[Lulu] of perro
(patas 4)
(raza caniche)
(dueno "Juan")
(ident gen1)
CLIPS> (send [Milu] get-patas)
4
CLIPS> (send [Milu] put-patas 3)
3
```



Plantillas

- Son como clases pero sin herencia
- Permiten representar hechos no ordenados
- Se definen como hechos (deffacts/assert)



Ejemplo (plantillas)

(deftemplate persona

```
(slot nombre (type SYMBOL))
  (slot edad (type NUMBER)
        (range 0 99)(default 20))
  (slot estado (type SYMBOL)
        (allowed-symbols soltero casado viudo)
        (default soltero))
)
(deffacts personas
        (persona (nombre Pepe))
        (persona (nombre Juan)(edad 25))
)
```

(defrule casa-mayores-25

```
?p <-(persona
    (nombre ?nombre)
    (edad ?edad)
    (estado soltero))
  (test (>= ?edad 25))
  (modify ?p (estado casado))
  (printout t ?nombre " tiene "
  ?edad " anios" crlf)
CLIPS> (reset)
Juan tiene 25 anios
CLIPS> (facts)
f-0 (initial-fact) CF 1.00
f-1 (persona (nombre Pepe) (edad
20) (estado soltero)) CF 1.00
f-3 (persona (nombre Juan) (edad
25) (estado casado)) CF 1.00
```



Módulos

- Los módulos representan diferentes estados en la resolución del problema y aíslan las clases y reglas
- Cada módulo tiene su propia agenda y debe indicar qué construcciones importa y exporta
- El hecho initial-fact debe ser importado del módulo MAIN
- Deberemos especificar a qué módulo pertenenen las construcciones con <módulo>::...
- Los módulos se fijan con focus. Ej. (focus A B C).
 Pasamos a otro módulo cuando la agenda está vacía
- También se puede cambiar de módulo con reglas, definiendo (declare (autofocus TRUE))



CLIPS integrado

```
#include <stdio.h>
#include "clips.h"
main() {
    InitializeCLIPS();
    Load("programa.clp");
    Reset();
    Run(-1L);
}
```



Jess (Java Expert System Shell)

 CLIPS reprogramado en Java, con ligeras variaciones, e integrado en el lenguaje

