

El lenguaje CLIPS

Javier Béjar

Curso 2021/2022

CS - FIB



El sistema CLIPS

- ⊙ CLIPS es un entorno para desarrollar sistemas expertos
- ⊙ Este define un lenguaje que permite la representación de conocimiento declarativo y procedimental
- ⊙ Su lenguaje permite representar reglas de producción y frames
- ⊙ Su base es un motor de inferencias con razonamiento hacia adelante
- ⊙ El motor de inferencias esta implementado sobre un intérprete del lenguaje

- ⊙ El lenguaje CLIPS deriva su sintaxis del lenguaje LISP
- ⊙ Se trata de un lenguaje parentizado con notación prefija
- ⊙ Los tipos de datos predefinidos que nos interesarán son: reales, enteros, strings, símbolos, apuntador a hechos, nombre de instancia y apuntador a instancia.
- ⊙ Los tipos habituales poseen los operadores mas comunes
- ⊙ El lenguaje de CLIPS auna tres paradigmas de programación: lenguaje de reglas, lenguaje funcional, lenguaje orientado a objetos

El lenguaje de reglas

- ⊙ Los dos elementos que permiten representar problemas utilizando reglas de producción son los hechos y las reglas.
- ⊙ Los hechos en CLIPS pueden ser de dos tipos *ordered facts* y *deftemplate facts*
- ⊙ Los Ordered Facts tienen formato libre, por lo tanto no tienen una estructura predefinida, siguen el esquema:

(*relacion* p1 p2 ... pn)

- ⊙ *relación* ha de ser un símbolo, el resto de parámetros puede ser de cualquier tipo, por ejemplo

(*padre* juan pedro)

(*num-hijos* juan 2)

- ⊙ Los deftemplate facts tienen una estructura predefinida, podrían asimilarse a representaciones al estilo de los frames.
- ⊙ Definimos una serie de campos (*slots*). Cada campo puede tener una serie de restricciones como tipo, cardinalidad y un valor por defecto (constante o función para calcularlo)

```
(deftemplate nombre-template "comentario"  
  (slot nombre-slot)  
  (multislot nombre-slot))
```

- ⊙ Por ejemplo:

```
(deftemplate persona  
  (slot nombre (type STRING))  
  (slot edad (type INTEGER) (default 0)))
```

- ⊙ La creación de hechos se realiza mediante la sentencia `assert` (uno solo) o `defacts` (un conjunto), por ejemplo:

```
(assert (padre pepe juan))  
(assert (persona (nombre "pedro") (edad 25)))  
(defacts mis-hechos  
  (casa roja) (pelota verde)  
  (persona (nombre "luis") (edad 33)))
```


- ⊙ (`facts`) permite saber que hechos hay definidos
- ⊙ (`clear`) borra todos los hechos definidos
- ⊙ (`retract <indice-hecho>`) elimina el hecho identificado por el índice dado
- ⊙ (`get-deftemplate-list`) retorna la lista de deftemplates definidos

- ⊙ Las reglas en CLIPS estan formadas por:
 - Una parte izquierda (LHS) que define las condiciones a cumplir
 - Una parte derecha (RHS) que define las acciones a realizar
- ⊙ Sintaxis:

```
(defrule nombre-regla "comentario"  
  (condicion-1) (condicion-2) ...  
  =>  
  (accion-1) (accion-2) ...)
```

- ⊙ Las variables definen patrones en las condiciones de las reglas
- ⊙ Se denotan poniendo un interrogante delante del nombre (`?variable`)
- ⊙ Existen variables anónimas (no importa su valor) para un valor `?` o para múltiples valores `$?`
- ⊙ Durante la ejecución se instanciarán las variables de las reglas con valores que permitan cumplir sus condiciones
- ⊙ Las variables de las reglas son locales, si queremos definir variables globales debemos usar la construcción `defglobal` (las variables globales se denotan `?*variable*`)

- ⊙ Tipos de condiciones en LHS
 - **Patrones** constantes, con variables o con wildcards: se instancian directamente con hechos en la base de hechos
 - Expresiones `not`, `and`, `or`, `exist` y `forall` con patrones
 - Tests de expresiones sobre las variables vinculadas (`test`)
- ⊙ Los **patrones** indican qué tipo de hechos deben instanciar las reglas, estos se establecen a través de restricciones sobre variables o valores constantes
- ⊙ Estas restricciones se pueden combinar mediante conectivas lógicas `~` (no), `&` (y) y `|` (o)
- ⊙ Se pueden usar condiciones complejas precedidas de :

- ⊙ Persona mayor de 18 años: `(persona (edad ?x&:(> ?x 18)))`
- ⊙ Persona de nombre juan o pedro: `(persona (nombre juan|pedro))`
- ⊙ Dos personas con nombres diferentes:
`(persona (nombre ?x)) (persona (nombre ?y&~?x))`
- ⊙ Nadie se llama pedro: `(not (persona (nombre pedro)))`
- ⊙ Todo el mundo es mayor de edad:
`(forall (persona (nombre ?n)
 (edad ?x)) (test (> ?x 18)))`

- Podemos obtener la dirección del hecho que instancia un patrón mediante el operador `<-`, por ejemplo:

```
(defrule mi-regla
  ?x <- (persona (nombre juan))
  =>
  (retract ?x)
)
```

- En la parte derecha de las reglas podemos poner cualquier sentencia válida en clips (ver manual)

- ⊙ Las reglas de CLIPS se organizan en módulos
- ⊙ Permiten estructurar el conocimiento y poder focalizar la ejecución de las reglas según su objetivo
- ⊙ La definición de un módulo se realiza mediante

```
(defmodule <nombre> "comentario" <export-import>)
```
- ⊙ Nada de lo definido en un módulo es visible salvo que lo exportemos
- ⊙ Para utilizar construcciones de otro módulo también tenemos que importarlas explícitamente
- ⊙ Existe un módulo por defecto llamado **MAIN** al que pertenece todo lo no definido en otro módulo

- ⊙ La exportación de construcciones de un módulo se realiza incluyendo la sentencia `export` en su definición. Podemos exportar cualquier cosa que definamos, por ejemplo:

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

- ⊙ La importación de construcciones a un módulo se realiza incluyendo la sentencia `import` en su definición. Podemos importar cualquier cosa visible que este definida en otro módulo, por ejemplo:

```
(defmodule B (import A deftemplate cubo))
```


- Podemos restringir qué módulos se usan para la ejecución de reglas mediante la sentencia (`focus <módulo>*`)
- Esta sentencia se puede incluir en la parte derecha de una regla para poder cambiar explícitamente de módulo
- Se puede hacer que la ejecución se focalice en el módulo de la última regla ejecutada declarando la propiedad `auto-focus` en una regla, por ejemplo:

```
(defrule JUAN::mi-regla
  (declare (auto-focus TRUE))
  (persona (nombre juan))
  => ...)
```

El intérprete de reglas tiene definidas unas estrategias de resolución de conflicto

- ⊙ **Profundidad**, las nuevas activaciones pasan al principio
- ⊙ **Anchura**, las nuevas activaciones pasan al final
- ⊙ **Simplicidad**, ante la misma posibilidad de activar, se prefiere las menos específicas (especificidad medida respecto a la complejidad de las condiciones)
- ⊙ **Complejidad**, tienen preferencia las reglas más específicas

- ⊙ **Estrategia LEX**, recencia de los hechos instanciados, tomando los hechos instanciados ordenadamente en cada regla y siguiendo orden lexicográfico de recencia
- ⊙ **Estrategia MEA**, Se ordenan por recencia respecto al hecho que instancia la primera condición, en caso de empate se sigue la estrategia LEX
- ⊙ **Aleatoria**, se disparan las reglas en orden aleatorio

El lenguaje funcional de CLIPS

- ⊙ CLIPS incluye un lenguaje de programación funcional
- ⊙ Éste permite definir nuevas funciones o programar las acciones a realizar en la parte derecha de las reglas
- ⊙ Toda sentencia o estructura de control es una función que recibe unos parámetros y retorna un resultado (paradigma funcional)

- ⊙ Asignación a una variable, retorna el valor asignado

`(bind <var> <valor>)`

- ⊙ Sentencia alternativa, retorna el valor de la última acción evaluada

`(if <exp> then <accion>* [else <accion>*])`

- ⊙ Bucle condicional, retorna falso, excepto si hay una sentencia de retorno que rompa el bucle

`(while <exp> do <accion>*)`

- ⊙ Bucle sobre un rango de valores, retorna falso, excepto si hay una sentencia de retorno

`(loop-for-count (<var> <v-i> <v-f>) do <accion>*)`

- ⊙ Ejecuta un conjunto de sentencias secuencialmente, retorna el valor de la última
`(progn <accion>*)`
- ⊙ Romper la ejecución de la estructura de control retornando el valor de la expresión
`(return <expr>)`
- ⊙ Romper la ejecución de una estructura de control
`(break)`
- ⊙ Alternativa caso, cada case se compara con el valor evaluado. Retorna la última expresión o falso si ningun case se cumple
`(switch <expr> (case (<comp>) then <accion>*)*
[(default <accion>*)])`

- ⊙ La construcción `deffunction` permite definir nuevas funciones

```
(deffunction <nombre> "Comentario"  
  (<?parametro>* [<$?parametro-wilcard>])  
  <accion>*)
```

- ⊙ La lista de parámetros puede ser variable, el parametro wilcard incluye en una lista el resto de parámetros
- ⊙ La función retorna la última expresión evaluada

Orientación a objetos en CLIPS

- ⊙ El lenguaje orientado a objetos de CLIPS permite representar la estructura del conocimiento
- ⊙ Se puede considerar como una extensión del constructor `deftemplate` que pretende completar la posibilidad de usar *frames* como herramienta de representación
- ⊙ Podemos definir clases como en los lenguajes orientados a objetos con slots y métodos
- ⊙ CLIPS tiene definido un conjunto inicial de clases que organizan los tipos predefinidos de CLIPS estableciendo una jerarquía entre ellos

- ⊙ La sentencia que permite definir una clase es `defclass`
- ⊙ Para definir una clase hay que especificar:
 1. El nombre de la clase
 2. Una lista de sus superclases (heredará de estas sus slots y métodos)
 3. Declaración de si es una clase abstracta o no (permitimos definir instancias)
 4. Si permitimos que instancias de esta clase puedan vincularse a patrones en la LHS de una regla
 5. Definición de los slots de la clase (slot, multi-slot)
- ⊙ Toda clase debe tener como mínimo una superclase

```
(defclass ser-vivo
  (is-a USER)
  (role abstract)
  (pattern-match non-reactive)
  (slot respira (default si)))
```

```
(defclass persona
  (is-a ser-vivo)
  (role concrete)
  (pattern-match reactive)
  (slot nombre))
```

La definición de slots incluye nuevas propiedades:

- ⊙ `(default ?DERIVE|?NONE|<exp>*)`
- ⊙ `(default-dynamic <expr>*)`
- ⊙ `(access read-write|read-only|initialize-only)`
- ⊙ `(propagation inherit|no-inherit)`
- ⊙ `(visibility public|private)`
- ⊙ `(create-accessor ?NONE|read|write|read-write)`
- ⊙ También se puede declarar el tipo, cardinalidad, ...

- ⊙ `make-instance` crea instancias de una clase
- ⊙ Al crear una instancia damos valor a sus slots , por ej:

```
(make-instance juan of persona (nombre "juan"))
```
- ⊙ Podemos crear conjuntos de instancias con la sentencia `definstances`, por ej:

```
(definstances personas  
  (juan of persona (nombre "juan"))  
  (maria of persona (nombre "maria"))  
)
```

- ⊙ La interacción con los objetos se realiza mediante **mensajes**
- ⊙ Estos mensajes tienen **manejadores** (*message handlers*) que los procesan y realizan la tarea indicada
- ⊙ Se definen mediante la sentencia `defmessage-handler`, su sintaxis es idéntica a la de las funciones.

```
(defmessage-handler <clase> nombre  
                  <tipo-h> (<param>*) <expr>*)
```

- ⊙ Existen diferentes tipos de manejadores pero nosotros solo los definiremos del tipo `primary`

- ⊙ Por defecto toda clase tiene definidos un conjunto de manejadores, por ejemplo:
`init`, `delete`, `print`
- ⊙ Al definir `create-accessor` en un slot estamos creando dos mensajes,
`get-nombre_slot`, `put-nombre_slot` para acceder y modificar el valor del slot
- ⊙ El acceso a los slots de un objeto dentro de un manejador se realiza mediante la variable `?self`, poniendo `:` delante del nombre del slot, por ejemplo:

```
(defmessage-handler persona escribe-nombre ()  
  (printout t "Nombre:" ?self:nombre crlf))
```


- ⊙ El envío de los mensajes se realiza mediante la sentencia `send`, el nombre de la instancia se pone entre corchetes, por ejemplo:

```
(send [juan] escribe-nombre)
```

```
(send [juan] set-nombre "pedro")
```

- ⊙ Los manejadores se pueden definir en cada clase, por lo tanto las subclases pueden ejecutar los manejadores de sus superclases. Para los de tipo `primary` estos se inician desde la clase más específica, si se quiere ejecutar los de las superclases se ha de usar la sentencia `call-next-handler`
- ⊙ Debe haber siempre como mínimo un manejador `primary` para cada mensaje

- ⊙ Para poder usar instancias en la RHS de una regla se utiliza la sentencia `object`, por ejemplo:

```
(defrule regla-personas
  (object (is-a persona) (nombre ?x))
  =>
  ...
)
```

- ⊙ La clase se ha de haber declarado como utilizable en la LHS de las reglas
- ⊙ La modificación de un slot de una instancia vuelve a permitir que se pueda volver a instanciar una regla con ella