Resumen CLIPS

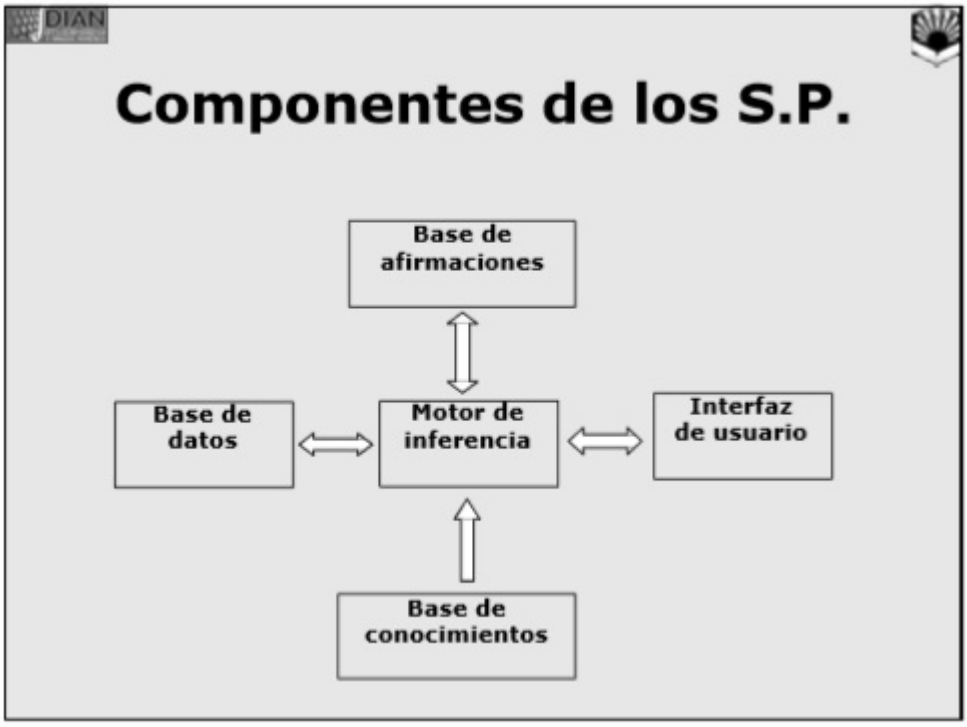
Tema 1. Introducción a los sistemas de producción

Sistemas de Producción

Las Reglas de Producción son reglas del tipo ***Si-Entonces***.

Las características de los Sistemas de producción:

- Se utilizan las reglas para examinar un conjunto de datos y solicitar nueva información hasta llegar a un diagnostico.
- También se denominan Sistemas basados en reglas



Componentes

- **Base de conocimientos**
  Todos los hombres son animales
  Todos los animales respiran
- **Base de afirmaciones**
  Juan es un hombre
- **Inferencia**
  Juan respira

Reglas de producción

La estructura general de las reglas son: ***Antecedente*** → ***Consecuente***. Donde:

- **Antecedente**: Contiene las claúsulas que deben de cumplirse para que la regla pueda ejecutarse
- **Consecuente**: Indica las conclusiones que se deducen de las premisas o las acciones que el sistema debe realizar cuando ejecuta la regla.

Inferencia

- Una regla se ejecuta (dispara) cuando se cumple su antecedente
- Las reglas se ejecutan hacia adelante: **Si se satisface el antecedente se efectúan las acciones del consecuente**
- Tipos de encadenamiento de reglas:
  - Encadenamiento hacia adelante o **basado en datos**.
  - Encadenamiento hacia detrás o basado en objetivos.

Control de Razonamiento
Se encarga de seleccionar una regla cuando hay varias disponibles. Métodos de resolución de conflictos:

- Ordenación de reglas
- Añadir nuevas cláusulas relacionadas con las inferencias
- Control mediante *Agenda*
- Conjunto de Reglas […]

Tema 2. Visión general de CLIPS

Componentes de CLIPS

- Interprete
- Interfaz interactivo
- Facilidades para la depuración
- Elementos de la Shell
  - Lista de hechos
  - Base de conocimientos
  - Motor de inferencia
- Dirigido por datos: Las reglas pueden emparejar con objeto y hechos.

Tipos de datos primitivos

- Entero (*integer*)
- Reales (*float*)
- Simbolo (*symbol*)
- Cadena (*string*)
- Direccion externa (*external-address*)
- Direccion de hecho (*fact-address*)
- Nombre de instancia
- Direccion de instancia (*instance-address*)

Otros conceptos

- **Campo**: Valor que puede tomar una variable
- **Tipos de campos dependiendo del valor que puedan tomar**:
  - Monocampo: Tipos de datos primitvos
  - Multicampo: Varios valores uni-campo
- **Constante**
- **Variable**

Ejemplos

- `?<nombre>` → monocampo (ámbito local)
- `$?<nombre>` → multicampo (ámbito local)
- `?*<nombre>*` → ambos (ámbito global)

Funciones

- **Lenguaje externo**

    - Funciones definidas por el usuario
    - Funciones definidas por el sistema

- **Funciones de CLIPS**

    - Funciones
    - Funciones genericas

- **Tipos**

    - Órdenes: Ejecutan una acción
    - Funciones: Devuelven un valor

- **Notación prefija para llamada**

    - ((nombrefuncion argumentos) → `(+(* 3 4) 8 )`)

Constructores

- **defmodule** → Define un módulo
- **defrule** → Define una regla
- **deffacts** → Define un hecho o conjunto de estos
- **deftemplate** → Define una plantilla
- **defglobal** → Define una variable global
- **deffunction** → Define una función
- **defclass** → Define una clase
- **definstances** → Define una instancia
- […]

Tema 3. Representación de Hechos en CLIPS

Representación de la información

Mediante:

- **Hechos**: Ordenados y no ordenados. Indice y dirección.
- **Objetos**: POO. Instancias de objetos.
- **Variables globales**: Constructor defglobal.

Hechos: Ordenes de uso

Órdenes de utilización de hechos:

- **assert** → Introduce datos en la base de hechos. *Se puede utilizar tambien deffacts*
- **facts** → Sirve para ver la base de hechos con formato `f-índice (hecho)`
- **undefacts** → Suprime los hechos insertados por una orden *deffacts*
- **ppdefacts** → Muestra la definición de un hecho definido por la orden *deffacts*
- **list-defacts** → Muestra los hechos definidos con *deffacts*
- **retract** → Elimina hechos de la base de hechos, se debe especificar el nombre o el indice del hecho se pueden eliminar varios hechos de golpe (retrat hecho1 hecho 2 ..) o eliminar todos con (retract *)
- **modify** → Modifica un hecho de la base de hechos
- **duplicate** → Duplica un hecho de la base de hechos
- **deftemplate** → Define una plantilla
- **ppdeftemplate** → Muestra una plantilla definida
- **undeftemplate** → Elimina una plantilla definida
- **deffacts** → Define un conjunto de hechos
- **reset** → Añade cada hecho especificado con *deffacts* en la lista de hechos o factlist. Tambien añade el hecho *initial-fact*.

Tambien dice que borra hechos e inserta hecho especial (?).

- **clear** → Limpia la base de hechos. Reinicializa el indice de hechos a 0 y elimina la base de conocimiento

Hechos: comandos

```
(assert <hecho>+)
(facts [<inicio> [<final> [máximo]]])
(retract <índice>+ |*)
(modify <índice> <nueva-casilla>+)
(duplicate <índice> <nueva-casilla>+)
  <nueva-casilla>::= (<nombre> <valor>)
```

Hechos: Tipos y Ejemplos

- **Ordenados** → `(casa calle-nueva 32)`
- **No ordenados (Realizados mediante plantillas)**: `(coche (marca ford)(modelo fiesta))`
  - El orden de los campos no es importante
  - Se pueden modificar utilizando las órdenes (modify) y (duplicate)

Hechos: Dirección

```
(defrule comenzar
    ?h <- (iniciar-programa)
  =>
    retract(?h)
    (printout t "Iniciando..." crlf)
)
```

Hechos: Ejemplos de plantillas

```
  (deftemplate nombre-plantilla
      (slot nombre)
      (multislot apellidos)
      (slot DNI)
  )
```

Tema 4. Hechos definidos a partir de plantillas

Sintaxis del constructor deftemplate

```
(deftemplate persona
  (slot nombre)
  (multislot apellidos)
  (slot DNI))
```

Y su respectivo hecho con el cual empareja

```
(persona (nombre Juan)(apellidos Perez Perez) (DNI 43765873B))
```

Propiedades de los slots

- default-attribute: puede tener la necesidad o no de que el hecho tenga que tener ese campo a huevos

    - ?NONE: no es necesario un argumento para ese campon
    - ?DERIVE: es obligatorio el argumento

```
(deftemplate dato
  (slot x (default ?NONE))
  (slot y (default ?DERIVE))
)
```

- type: reestriccion de tipo para los slots, es decir, dicho slot debe llevar una variable de un determinado tipo (SYMBOL, FLOAT, INTEGER, NUMBER, STRING, …)

```
(deftemplate dato
  (slot x (type STRING))
  (slot y (type NUMBER))
)
```

Dentro de esta reestriccion se puede incluir otra mas severa, acotando mas el rango

```
(deftemplate dato
  (slot x (allowed-symbols Pepe))
  (slot y (allowed-integer 6 5 9))
  (slot z (allowed-values "Ricardo" rico 99 1.e9))
)
```

Siendo posible acotar para los distintos tipos existentes en CLIPS. Siendo posible especificar rangos como se indica en el siguiente ejemplo

```
(deftemplate dato
  (slot x (allowed-symbols Pepe))
  (slot y (type integer)(range 8 90))
)
```

Tema 5. Reglas

El formato de una regla es *if-else*, es decir, si el antecedente es verdadero entonces se ejecuta el consecuente.

- Base de conocimiento: Conjunto de reglas que describen el problema a resolver
- Activación de reglas: Entidad patrón → hechos ordenados o plantillas, e instancias de clases
- Motor de inferencia: Comprueba el antecedente de las reglas y aplica el consecuente

Definición de reglas

Se realiza mediante el constructor defrule, y posee el siguiente formato:

```
(defrule regla1
  (frigorifico abierto)
  =>
  (comida mala)
)
```

- Consideraciones
    - En caso de que haya una regla con el mismo nombre, la última machaca a la anterior
    - No hay limite en el numero de elementos condicionales y acciones de una regla

- Puede no haber ningún elemento condicional en el antecedente y se usa automaticamente como elemento condicional
- Puede no haber ninguna acción en el consecuente, y la ejecución de la regla no tiene ninguna consecuencia

Ciclo de ejecución de reglas

- Las regla se ejecutan con el comando (run)
- Si se ha alcanzado el numero máximo de disparos, se detiene la ejecución
- Se actualiza la agenda según la lista de hechos
- Se selecciona la instancia de regla a ejecutar de acuerdo a prioridades y estrategia de resolución de conflictos
- Se dispara la instancia seleccionada, se incrementa numero de disparos y se elimina de la agenda
- Volver al paso 2

Sintaxis del antecedente

El antecedente está compuesto por una serie de elementos condicionales. Son un conjunto de reestricciones que se usan para verificar si se cumple un campo o slot de una entidad patrón.

- EC patron → [Definido Abajo]

- EC test → Comprueba el valor devuelto por una función, si se satisface si la función devuelve un valor distinto de **FALSE**. No se satisface si la función devuelve un valor **FALSE**

  - Ejemplo:
    `(defrule diferencia (dato ?x) (valor ?y) (test (>= (abs(- ?x ?y)) 3)) => )`

- EC and → Se satisface si se satisface todos de los EC que lo componen. Se pueden mezclar ands y ors en el antecedente.

  - Ejemplo:
    `(defrule posibles-desayunos (tengo zumo-natural) (or (and (tengo pan) (tengo aceite)) (and (tengo leche) (tengo cereales))) => (assert (desayuno sano)) )`

- EC or → Se satisface si se satisface cualquiera de los EC que lo componen, si se satisface mas de uno, la regla se activará varias veces.

  - Ejemplo:

```
(defrule posibles-desayunos
  (tengo pan) (or (tengo mantequilla)
  (tengo aceite))
  =>
(assert (desayuno tostadas))
```

- EC not → Se satisface si no se satisface el EC que contiene. Donde solo puede negarse una vez

  - Ejemplo:

```
(defrule movil-sin-bateria
  (not (queda bateria))
  (not (hay enchufe))
=>
  (assert (estado panico))
)
```

- EC exists → Permiten comprobar si una serie de ECs se satisface por algún conjunto de hechos.

  - Ejemplo:

```
(defrule dia-salvado
  (objetivo salvar-el-dia)
```

```
  (exists (heroe (estado desocupado)))
=>
  (printout t "El día está salvado" crlf))
```

- EC forall → Permite comprobar si un conjunto de EC se satisface para toda ocurrencia de otro EC especificado. Se satisface, para toda ocurrencia del primer EC, se satisfacen los demás ECs
  - Ejemplo:

```
(defrule todos-limpios
  (forall (estudiante ?nombre)
    (lengua ?nombre)
    (matematicas ?nombre)
    (historia ?nombre))
  =>)
```

- EC logical → Asegura el mantenimiento de verdad para hechos creados mediante reglas que usan EC Logical.
  - Los hechos del antecedente proporcionan soporte lógico a los hechos creados en el consecuente.
  - Un hecho puede recibir soporte lógico de varios conjuntos distintos de hechos.
  - Un hecho permanece mientras permanezca alguno de los que lo soportan lógicamente.
  - Los ECs incluidos en el EC logical están unidos por un *and* implícito.
  - Puede combinarse con ECs *and*, *or* y *not*
  - Ejemplo:

```
(defrule puedo-pasar
  (logical (semaforo verde))
=>
  (assert (puedo pasar))
)
```

Sintaxis del antecedente

Siendo el siguiente formato el obligatorio para realizar la comparacion

- Para hechos ordenados → (…)
- Para hechos no ordenados → (… ())

Las distintas reestricciones son:

- Literales → Define el valor exacto que debe de poseer el campo, solo posee constantes.

- Comodines → Indican que cualquier valor en esa posición en válido para emparejar con la regla
  - Comodín monocampo → ?
  - Comodín multicampo → $?

- Variables → Almacena el valor de un campo para despues utilizarlo en otro elementos condicionales o consecuentes de una regla
  - Cuando la variabel aparece por 1ª vez, actua como un comodin, pero el valor queda **ligado al valor del campo**

- Conectivas → Permiten unir reestricciones y variables, utilizando los conectores lógicos and/or/not (&/|/~ → Siendo la prioridad de las conectivas ~/&/|)
  - Excepción: Si la primera restricción es una variable seguida de la conectiva &, la primera reestricción (la variable) se trata como una reestricción aparte
  - Ejemplo: ?x&rojo|azul equivale a ?x&(rojo|azul)

- Predicado → Se restringe un campo según el valor de verdad de una expresión lógica.
    - Esta se indica mediante dos puntos (:) seguidos de una llamada a función predicado.
    - La restricción se satisface si la función devuelve un valor dto. *FALSE*
    - Normalmente se usan junto a una restricción conectiva y a una variable
    - Ejemplo: ::= | … | :

```
(defrule predicado1
  (datos ?x&:(numberp ?x))
  =>
)
```

    - Las funciones predicado que proporciona CLIPS son:
    - (evenp) → Comprueba que el argumento sea par
    - (floatp) → Comprueba que el argumento sea FLOAT
    - (integerp) → Comprueba que el argumento sea de tipo INTEGER
    - (numberp) → Comprueba que el argumento sea un numero
    - (oddp) → Comprueba que el argumento sea impar
    - (stringp) → Comprueba que el argumento sea de tipo STRING
    - (symbolp) → Comprueba que el argumento sea de tipo SYMBOL
    - Y las funciones de comparación:
    - (eq +)
    - (neq +)
    - (= +)
    - (<> +)
    - (> +)
    - (< +)
    - (>= +)
    - (<= +)

- Valores devueltos → Se usa el valor devuelto por una función para restringir un campo. Cuyo valor devuelto debe ser de uno de los tipos primitivos de datos y se sitúa en el patrón como si tratase de una restricción literal en las comparaciones.
    - Se implica mediante el caracter "="

```
(deftemplate datos
(slot x)(slot y)
)

(defrule triple
(datos (x ?x) (y =(* 3 ?x)))
=>
)
```

- Direcciones de hechos: Para realizar modificaciones, duplicaciones o eliminaciones de hechos en el consecuente de una regla. Es necesario que en la regla se obtenga el indice del hecho sobre el que se desea actuar.
    - Ejemplo:

```
(defrule triple
(datos (x ?x) (y =(* 3 ?x)))
=>
)
```

Propiedades de una regla

- Los comandos disponibles para el manejo de reglas son:

- (ppdefrule) → Visualiza una regla por pantalla
- (list-defrules |*) → Muestra las reglas que hay
- (rules) →
- (undefrule |*) → Elimina la regla

- La declaración de propiedades se incluye tras el comentario y antes del antecedente
- Se indica mediante la palabra reservada *declare*
- Una regla puede tener una única sentencia *declare*
  - Prioridad
  - Se indica en la declaración de propiedades con la palabra reservada *salience*.
  - Puede tomar valores [-10000,10000], teniendo el valor 0 por defecto.
  - La prioridad puede evaluarse cuando se define la regla (por defecto), cuando se activa y en cada ciclo de ejecución.
  - Ejemplo:
    `(defrule triple (declare (salience 1000)) (datos (x ?x) (y =(* 3 ?x))) => )`

Tema 7. Variables globales

- El formato es → ?*nombre*, y se definen con *defglobal*
- Se pueden acceder desde dentro de los patrones
- El comando **bind** asigna un valor a una variable.
- El comando **reset** reinicializa su valor.
- Para eliminarlas se puede utilizar el comando **clear** o **undefglobal**
- Ejemplo:
  `(defglobal ?*x* = 3 ?*y* = 6 )`
- Diferencias con las variables locales:
  - No se puede usar de igual forma en el antecedente de una regla al asociarle un valor.
  - Ejemplo:

```
(defrule ejemplo ;incorrecto
(fact ?*x*)
=> )

(defrule ejemplo ;correcto
(fact ?y&:(= ?y ?*x*))
=>
(bind ?*x* 3))
```

Tema 8. Modulos

- Con **defmodule** agrupa constructores (hechos,reglas,etc.) en módulos
- Los módulos se pueden borrar con la orden clear (y siempre debe de haber un módulo MAIN)
- Los módulos asociados con un nombre pueden especificarse explicitamente (*modulo::elemento*) o implicitamente (modulo actual o modulo activo).
- Con el comando **set-current-module** se activa un módulo diferente
- Los elementos deben de ser exportados o importados para poder ser usados por otros módulos

Ejecución de reglas

- Las formas de alterar el orden normal son:
  - Con la orden **focus** en el consecuente de una regla. Incluye un nuevo módulo en la pila de módulos activos y pasa el control a la agenda de este módulo.
  - Con la orden **return** en el consecuente de una regla. Esto provocaría que se pasaría como módulo activo el siguiente módulo activo el siguiente módulo de la pila de módulos activos y pasa el control a su agenda.

INLUIR ALGUN EJEMPLILLO

Tema 9. Funciones

Con el constructor deffunction permite crear nuevas funciones dentro de CLIPS, compuesta por:

- Nombre
- Comentario (opcional)
- Cero o mas parametro regulares
- Un parametro comodin opcional que gestiona un número variable de argumentos
- Una secuencia de acciones, o expresiones, que se ejecutaran de forma secuencial cuando se llame a la función definida mediante este constructor.
- El formato es el siguiente:

```
(deffunction <nombre> [<comentario>]
  (<parametro-regular>*
  [<parametro-comodin>])
<accion>*

===================================

<paramatro-regular> ::= <variable-de-campo-simple>

===================================

<paramatro-comodin> ::= <variable-multicampo>
```

- Conceptos:
  - Una función debe de tener un nombre único
  - Una función debe de ser declarada antes de ser llamada
  - Tambien pueden llamarse a si misma y ser recursiva

Tema 11. Acciones y funciones

- Funciones de predicado:
  - (integerp)
  - (floatp)
  - (numberp)
  - (symbolp)
  - (stringp)
  - (lexemep)
  - (evenp)
  - (oddp)
  - (multifieldp)

- Otras funciones de predicado:
  - (eq +)
  - (neq +)
  - (= +)
  - (<> +)
  - (< +)
  - (<= +)
  - (> +)
  - (>= +)
- Y venga funciones de predicado, que son gratis:
  - (and +)
  - (or +)
  - (not)
- Funciones multicampo:
  - (create$ *)
  - (length$)
  - (nth$)
  - (member$)
  - (subsetp)
  - (subseq$)
  - (first$)
  - (rest$)
  - (explode$)
  - (implode$
  - (insert$ +)
  - (replace$ +)
  - (delete$
- Funciones de cadena:
  - (str-cat *)
  - (sym-cat *)
  - (str-length)
  - (sub-string)
  - (str-compare)
- Funciones matemáticas
  - (+ +)
  - (- +)
  - (* +)
  - (/ +)
  - (div +)
  - (mod)
  - (sqrt)
  - (**)
  - (round)
  - (abs)
  - (max +)
  - (min +
- Funciones procedurales

- (bind *)
- *if-esle*

```
(if <expresión>
  then <acción>*
  [else <acción>\*])
```

- *switch-case*
  (sw itch

  [])

::=
(case then)

::=
(default)

- Bucles

- (w hile [do] *)

- Bucle raro:

```
(loop-for-count <rango> [do] <acción>\*)
<rango> ::=
  (<variable> <índice-final>) |
  (<variable> <índice-inicio> <índice-final>)

<índice-final> | <índice-inicio> ::= <expr-entera>
<índice-final> ::= <expr-entera>
(return [<expresion>])
(break)
```