



CUADERNO DE PRACTICAS

Sistemas Inteligentes

Práctica 6

Funciones definidas por el sistema

David Sánchez Fernández

i22safed@uco.es

INDICE

1. Conocimiento y representación del conocimiento en CLIPS	2
1.1. Resumen de la practica	2
1.2. Ejemplos de clase	5
1.2.1. Ejemplo 1	5
1.2.2. Ejemplo 2	5
1.3. Ejemplos propuestos	6
1.3.1. Ejemplo 1	7
1.3.2. Ejemplo 2	8
2. Representación de la información mediante hechos	9
2.1. Introducción a las plantillas	9
2.2. Estructura de plantillas	9
2.2.1. Ejemplo de plantilla	10
2.3. Ordenes relacionadas con plantillas	10
2.4. Constructor de hechos (deffacts)	11
2.5. Estructura de constructor de hechos	11
2.5.1. Ejemplo del constructor de hechos	11
2.6. Ejercicios de plantillas	11
3. Representación del conocimiento mediante reglas	16
3.1. Introducción	17
3.2. Consideraciones	18
3.3. Sintaxis del antecedente	18
3.4. Propiedades de una Regla	30
3.4.1. Prioridad	30
3.5. Ejercicios propuestos	31
4. Funciones definidas por el sistema	50
4.1. Funciones de predicado	50
4.1.1. Funciones de tipo	50
4.1.2. Funciones aritméticas	55
4.1.3. Funciones lógicas	60

4.2 Funciones multicampo	62
4.3. Funciones de cadena	69
4.4. Funciones de E/S	71
4.5. Funciones matemáticas	74
4.6. Funciones procedurales	79
5. Bibliografía	80

1. Conocimiento y representación del conocimiento en CLIPS

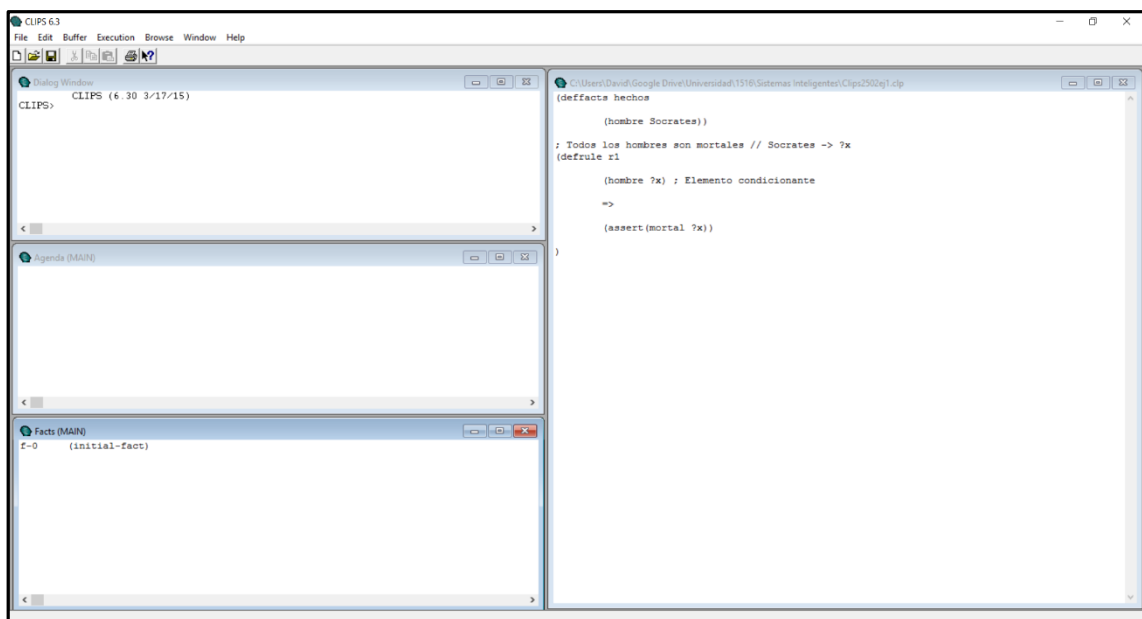
1.1. Resumen de la practica

En la primera práctica hemos realizado una toma de contacto con el lenguaje de programación CLIPS, un lenguaje basado en reglas y hechos destinado principalmente para sistemas expertos.

La arquitectura está compuesta principalmente por tres elementos:

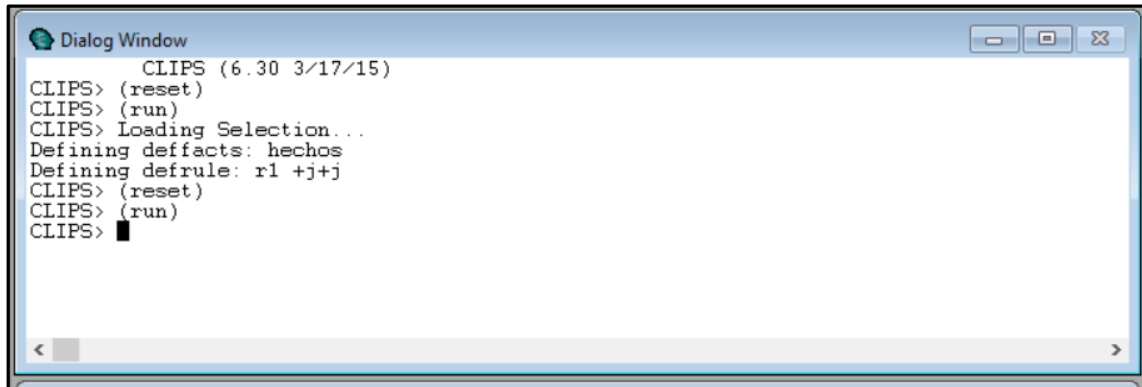
- Base de reglas: Almacena que representan el conocimiento para la resolución del problema.
- Interprete o motor de inferencia: Maneja la ejecución de las reglas del programa.
- Memoria del trabajo: Este elemento contiene los hechos que representan el conocimiento que el sistema ha adquirido del problema que se intenta recuperar.

El entorno con el que trabajaremos a lo largo de las practicas será CLIPS en su versión 6.3, que tiene el siguiente aspecto:

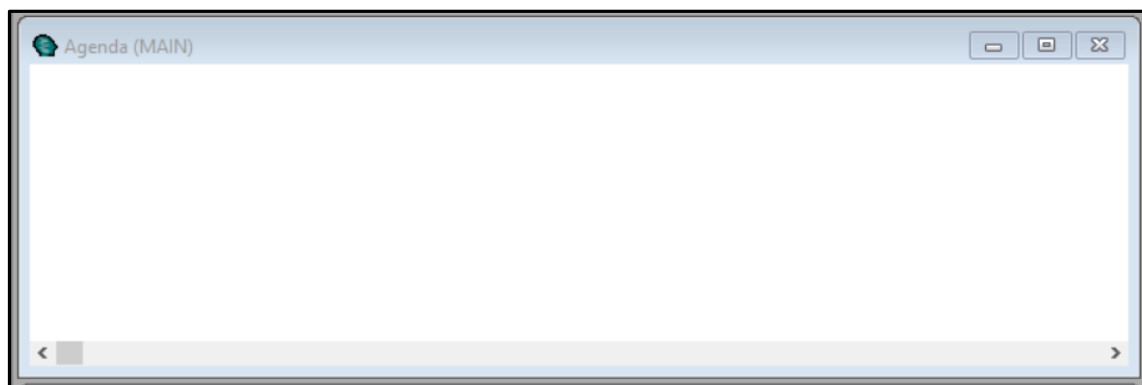


Las ventanas que más utilizaremos serán las siguientes

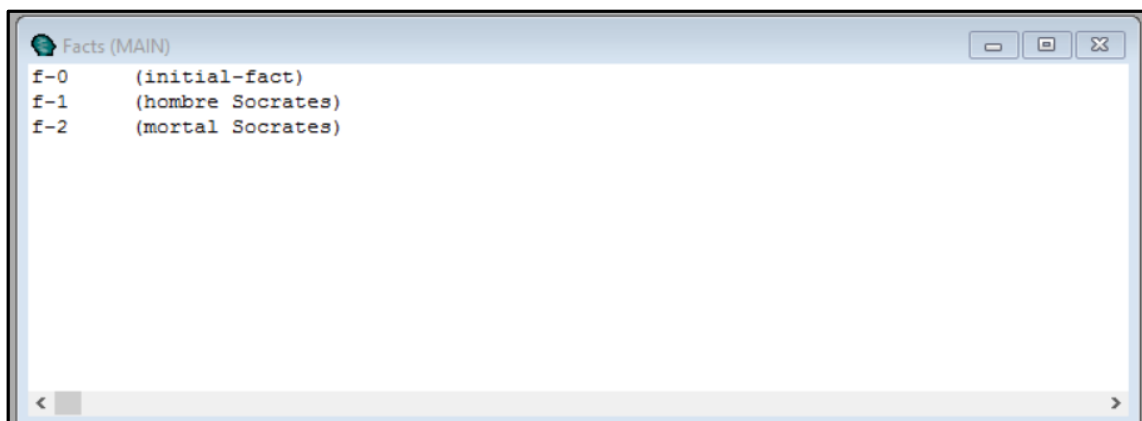
- Ventana de dialogo: Podríamos definir esta ventana como consola donde podremos ver los pasos que se siguen durante la ejecución e introducir órdenes



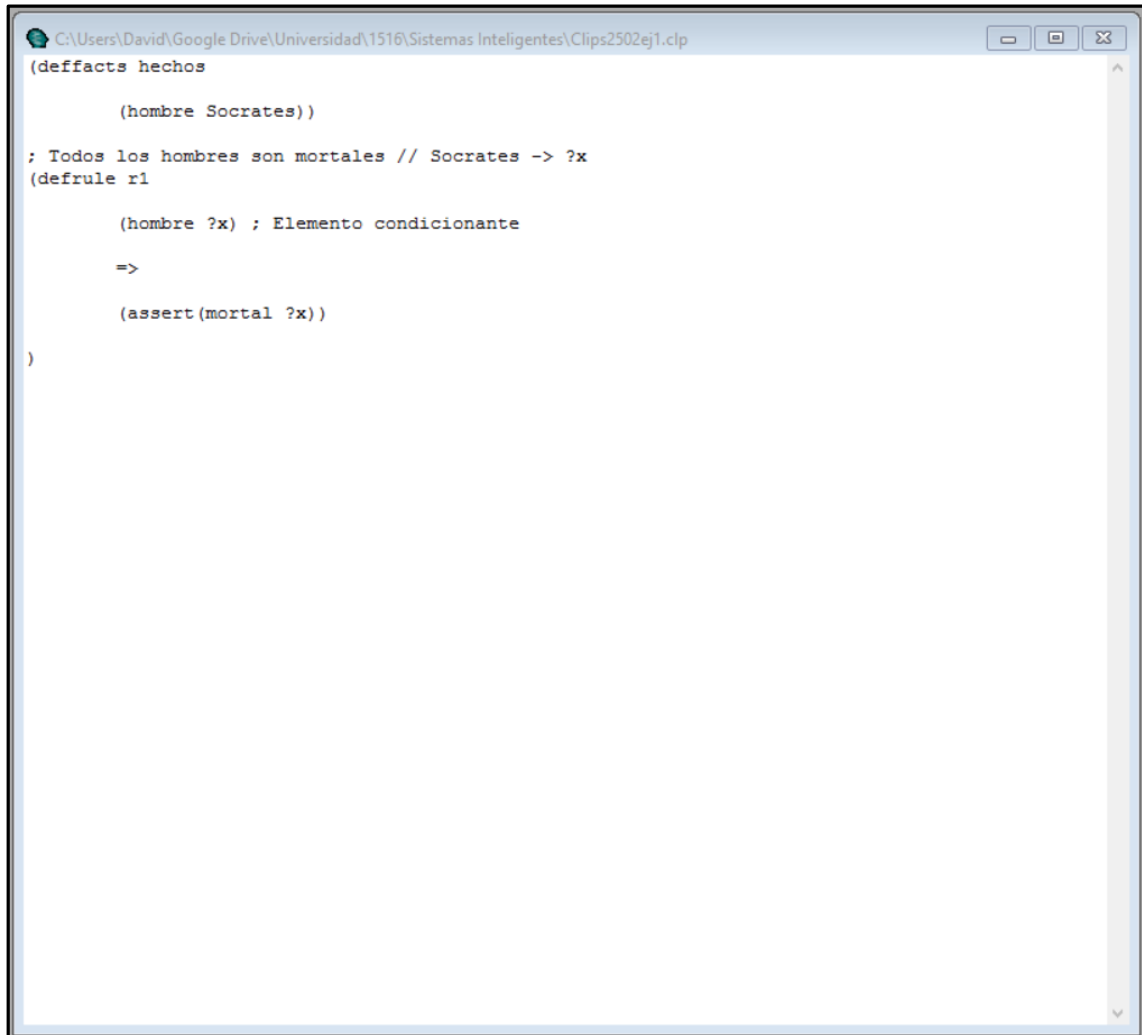
- Agenda: Aparece la colección de reglas activadas



- Base de hechos: En esta ventana aparecen los hechos definidos



- Editor: Utilizaremos el editor para visualizar el código, modificarlo y cargarlo (seleccionar texto y pulsar Ctrl. + K)



The image shows a screenshot of a CLIPS editor window. The title bar indicates the file path: C:\Users\David\Google Drive\Universidad\1516\Sistemas Inteligentes\Clips2502ej1.clp. The editor contains the following code:

```
(deffacts hechos

  (hombre Socrates))

; Todos los hombres son mortales // Socrates -> ?x
(defrule r1

  (hombre ?x) ; Elemento condicionante

  =>

  (assert(mortal ?x))

)
```

Para ejecutar un programa, primero debemos cargar el programa en el buffer pulsando Ctrl + K (Load selection) o pulsando en el menú de buffer las opciones de Load buffer.

Una vez cargado, ahora en el menú Execution, debemos pulsar Reset (Ctrl + E), y acto seguido run (Ctrl + R)

1.2. Ejemplos de clase

1.2.1. Ejemplo 1

```
(deffacts hechos
  (hombre Socrates))

; Por comparación, hombre empareja hombre, y por ende Sócrates
; se asigna a ?x
(defrule r1
  (hombre ?x) ; Elemento condicionante
=>
  (assert(mortal ?x))
)
```

En este ejemplo comprobamos si **Socrates es mortal**, por lo tanto, declaramos que **Socrates es hombre**. Después definimos la regla **todos los hombres son mortales** además declaramos **hombre** como elemento condicionante. Cuando la regla valide el elemento condicionante, el aserto se activará confirmando que **Socrates es mortal**.

1.2.2. Ejemplo 2

```
(deffacts h1      ; constructor de hechos
(hombre Socrates); hecho ordenador con dos campos ocupados por
                  ; símbolos
(hombre Platon)
)
(defrule r1      ; declaración de la primera regla
  (hombre ?x); Elemento condicional del patrón
=>
  (assert(mortal ?x)); Acción de afirmación del hecho
```

```

; ordenado (mortal)

)

(defrule r2 ; Declaración de la segunda regla
  (mortal ?x)
  =>
  (printout t ?x " Es mortal" crlf )
)

```

Este ejemplo es similar al anterior con la única diferencia de que tenemos una regla, es decir, afirmamos que Socrates es mortal y Platon es mortal.

Comprobamos primeramente el antecedente de la primera regla, hombre coincide con hombre y a la variable del consecuente (?x) se le asigna el valor Socrates después ejecutamos la segunda regla y donde mortal coincide y por lo tanto imprime Socrates (?x) es mortal.

Pasa lo mismo para el caso de Platón

1.3. Ejemplos propuestos

1.3.1. Ejemplo 1

```

; Ejemplo para ver como funciona el motor de inferencia
; - Como el motor de inferencia realiza el mecanismo de inferencia
; de comparación de patrones
; - Como se liga el valor de un campo a una variable
; - Ver elemento condicional del test
; - Ver operadores aritmeticos
; - Ver como se liga la direccion de un hecho a una variable
; - Ver como retractar un hecho

; =====
; Presentar por pantalla los diez primeros números naturales

```



```

( deffacts h1; constructor de hechos
    (n 0) ; Hecho ordenado
)

(defrule r1
    ?f<-(n ?x) ; Elemento condicional de patron(ECP)
                ; A la variable ?x se le ligará valores de los
                ; hechos que emparejen
                ; A la variable ?f se le liga la direccion de hecho
                ; con el que empareje el ECP

    (test ( z ?x 10)) ; Elemento condicional del test

    =>

    (printout t "n= " ?x crlf) ; Accion de imprimir

    (assert (n (+ ?x 1)))      ; Afirmación de un hecho nuevo ( n
                                ; resultado_de_la_suma)

    (retract ?f)               ; Elimina el hecho de cuya dirección está en
                                ; la variable ?f

)

```

1.3.2. Ejemplo 2

```
; Ejemplo de como sumar los 7 numeros naturales

( deffacts h1 ; Constructor de hechos
  (n 0)      ; Hecho ordenado
  (suma 0)

)

(defrule r1
?f1<-(n ?x) ; Elemento condicional patron (ECP)
      ; A la variable ?x se le ligará los valores de
      ; los hechos emparejados
      ; A la variable ?f se le liga la direccion de
      ; hecho con el que empareje el ECP

?f2<-(suma ?s)
(test (< ?x 7))      ; Elemento condicionante del test
=>
(printout t "n= " ?x crlf)      ; Accion de imprimir

      (assert (n (+ ?x 1)))      ; Afirmacion de un hecho
      ; nuevo ( n resultado-
      ; de-la-suma)

      (retract ?f1)      ; Elimina el hecho cuya
      ; direccion esta en la
      ; variable ?f1
(retract ?f2)      ; Elimina el hecho cuya
      ; direccion esta en la
```

```
                ; variable ?f2

(assert (suma (+ ?s ?x)))
)
```

2. Representación de la información mediante hechos

2.1. Introducción a las plantillas

Utilizaremos el constructor de plantillas (deftemplate)

```
(deftemplate persona
  (slot nombre)
  (slot edad)
  (multislot direccion)
)
```

2.2. Estructura de plantillas

· Valores por defecto

```
<atributo_por_defecto> ::=
(default ?DERIVE | ?NONE | <expresion>*) | ...
```

2.2.1. Ejemplo de plantilla

Ejemplo del cuestionario el de los slots w, x, y, z

```
(slot edad ( type LEXEME)) ; tipo lexema == string
(slot edad ( type INTEGER SYMBOL)) ; tipo entero o
                                ; símbolo
(slot x(allowed-values uno dos)) ; valores permitidos
                                ; uno o dos,
                                ; es decir que no puede
                                ; introducir 1 ó 2
```

2.3. Ordenes relacionadas con plantillas

```
(ppdeftemplate <nombre-plantilla>) ; Muestra plantilla ya ;definida
(list-deftemplates [<nombre-modulo> | *]) ; Lista las plantillas que
                                           ; están siendo utilizadas
                                           ; por el sistema
(undeftemplate <nombre-plantilla> | *) ; Elimina una plantilla
                                           ; existente
```

2.4. Constructor de hechos (deffacts)

Corchetes: Opcional (list-deftemplates [<nombre-modulo> | *])

Asterisco: 0 ó más hechos (list-deftemplates [<nombre-modulo> | *])

Suma: Que se repite (eq <expression> <expression>+)

2.5. Estructura de constructor de hechos

```
(deffacts <nombre-definición> [<comentario>] <hecho> *)
```

2.5.1. Ejemplo del constructor de hechos

```
(deffacts personas
  (persona (nombre Juan)
    (apellidos Fernandez)
    (dni 1))
)
```

2.6. Ejercicios de plantillas

; Ejercicio 1

```
(deftemplate persona

  (slot nombre (type STRING))
  (slot apellido (type STRING))
  (slot color_ojos (type SYMBOL) (default marrones))
  (slot altura (type FLOAT) (default 1.85))
  (slot edad (type INTEGER) (default 22))

)
```

; Ejercicio 2

```
(deftemplate pacientes
  (slot nombre)
  (slot apellido)
  (slot dni)
  (slot seguroMedico)
)
```

```
(deftemplate visitas
  (slot fecha)
  (multislot sintomas)
  (slot pruebas)
  (slot medicacion)
  (slot dni)
)
```

; Ejercicio 3

```
(deftemplate trayectos_aereos
  (slot origen)
  (slot destino)
)

;v1
(assert(trayectos_aereos(origen Lisboa)(destino Paris)))
(assert(trayectos_aereos(origen Estocolmo)(destino Paris)))
(assert(trayectos_aereos(origen Lisboa)(destino Madrid)))
(assert(trayectos_aereos(origen Roma)(destino Madrid)))
(assert(trayectos_aereos(origen Roma)(destino Lisboa)))
(assert(trayectos_aereos(origen Paris)(destino Roma)))
(assert(trayectos_aereos(origen Frankfurt)(destino Roma)))
(assert(trayectos_aereos(origen Roma)(destino Frankfurt)))
```

```

(assert(trayectos_aereos(origen Frankfurt)(destino Estocolmo)))

;v2
(deffacts datos
  (trayectos_aereos(origen Lisboa)(destino Paris))
  (trayectos_aereos(origen Estocolmo)(destino Paris))
  (trayectos_aereos(origen Lisboa)(destino Madrid))
  (trayectos_aereos(origen Roma)(destino Madrid))
  (trayectos_aereos(origen Roma)(destino Lisboa))
  (trayectos_aereos(origen Paris)(destino Roma))
  (trayectos_aereos(origen Frankfurt)(destino Roma))
  (trayectos_aereos(origen Roma)(destino Frankfurt))
  (trayectos_aereos(origen Frankfurt)(destino Estocolmo))
)

```

; Ejercicio 4

```

(deftemplate familia
  (slot familiar1(type STRING))
  (slot familiar2(type STRING))
  (slot relacion(type STRING))
)

(deffacts hechos
  (familia(familiar1 "Toni") (familiar2 "Elena") (relacion
"marido_mujer"))
  (familia(familiar1 "Toni") (familiar2 "Rafa") (relacion
"padre_hijo"))
)

```

```

(familia(familiar1 "Toni") (familiar2 "Saul") (relacion
"padre_hijo"))
(familia(familiar1 "Lourdes") (familiar2 "Rafa") (relacion
"madre_hijo"))
(familia(familiar1 "Lourdes") (familiar2 "Saul") (relacion
"madre_hijo"))
      (familia(familiar1 "Rafa") (familiar2 "Toni") (relacion
        "hermanos"))
)

```

; Ejercicio 5

```

(deftemplate libro
  (multislot autor (type STRING) (default ?NONE))
  (multislot titulo (type STRING) (default ?NONE))
  (multislot editorial (type STRING) (default ?DERIVE))
  (multislot edicion(type NUMBER) (default 0))
  (multislot anyo (type INTEGER) (default ?DERIVE))
)

(deffacts hechos
  (libro (autor "Mira,J.Delgado A.E, Boticario J.G") (titulo "Aspectos
basicos de la inteligencia artificail") (editorial "Sanz y
Torres")(edicion) (anyo 1995))

  (libro (autor "Galan, S. F. ;Boticario, J. G. ;Mira, J") (titulo
"Problemas resueltos de inteligencia artificial") (editorial "Adisson-
weley IberoAmericana")(edicion)(anyo 1998))

  (libro (autor "Rich E. Knight, K") (titulo "Inteligencia Artificial")
(editorial "McGraw-Hill") (edicion 2)(anyo 1994))

```



```
)
```

; Ejercicio 6

```
(deftemplate coche
```

```
  (slot num_coches(type INTEGER) (default ?DERIVE))
  (slot modelo(type STRING))
  (slot cilindrada(type INTEGER) (default ?DERIVE))
  (slot combustible(type STRING) (allowed-strings "gasoil"
    "gasolina"))
  (multislot num_puertas(type INTEGER))
  (multislot color(type STRING))
  (multislot vendedor(type STRING) (default ?DERIVE))
  (multislot fecha(type SYMBOL) (default ?DERIVE))
  (multislot cliente(type STRING))
```

```
)
```

```
(deffacts hechos
```

```
(coche(num_coches 1)(modelo "clio")(cilindrada 1600)(combustible
"gasolina")(num_puertas 3)(color "azul"))
(coche(num_coches 1)(modelo "clio")(cilindrada 1800)(combustible
"gasoil")(num_puertas 5)(color "blanco"))
(coche(num_coches 1)(modelo "megane")(cilindrada 1800)(combustible
"gasoil")(num_puertas 5)(color "dorado"))
(coche(num_coches 2)(modelo "megane")(cilindrada 1600)(combustible
"gasolina")(num_puertas 5)(color "gris"))
(coche(num_coches 1)(modelo "laguna")(cilindrada 2000)(combustible
"gasolina")(num_puertas 5)(color "negro"))
(coche(num_coches 1)(modelo "megane")(fecha 10/10/2003)(vendedor "Juan
Perez")(cliente "Esteban Losada"))
(coche(num_coches 1)(modelo "laguna")(fecha 13/10/2003)(vendedor "Ana
Ballester")(cliente "Juan Cano"))
(Juan Perez vendio un megane el 10/10/2003 al cliente Esteban Losada)
(Ana Ballester vendio un laguna el 13/10/2003 al cliente Juan Cano)
```

```

)

; Ejercicio 8

(deftemplate ingredientes
  (multislot pisto)
  (multislot tortilla)
  (multislot despensa)
  (multislot comprar)
)

(deffacts hechos
  (ingredientes (pisto pimientosVerdes pimientosRojos berenjenas
    calabacines cebollas tomateTriturado sal aceite))
  (ingredientes(tortilla huevos patatas cebollas sal aceite))
  (ingredientes(despensa pimientosVerdes pimientosRojos cebollas
    aceite)(comprar calabacines berenjenas tomateTriturado sal huevos
    patatas ))
  (compras)
  (ingredientes(despensa pimientosVerdes pimientosRojos cebollas aceite
    calabacines berenjenas tomateTriturado sal huevos patatas)(comprar
    todo comprado)))
)

(deffacts hechos2
  (cocinar)
  (ingredientes(despensa pimientosVerdes pimientosRojos aceite
    calabacines berenjenas sal huevos patatas)(comprar tomateTriturado
    cebollas)))
)

```

3. Representación del conocimiento mediante reglas

Presentar un resumen del [tema 6](#) destacando los aspectos más relevantes del mismo. Pueden ayudarse de cuadros esquemáticos y ejemplos.

Añadir a este resumen los ejercicios propuestos en clase acompañados de comentarios al código que expliquen y aclare su contenido

3.1. Introducción

En este tema vemos el concepto de reglas, cuya estructura está compuesta por un antecedente y un consecuente (*if-e/se*), es decir, si se cumple el antecedente (condición) según los hechos almacenados en la lista de hechos, entonces pueden realizarse las tareas (acciones) especificadas en el consecuente.

La estructura de la regla a nivel de código es ésta:

```
(defrule <nombre-regla>
  [<comentario>]
  [<propiedades>]
  <elemento-condicional>*
  =>
  <acción>*)
```

Un ejemplo de regla:

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

3.2. Consideraciones

- Una regla con el mismo nombre que otra, aun siendo errónea, machaca a la anterior.
- No hay límite en el número de elementos condicionales y acciones de una regla.
- Puede no haber ningún elemento condicional en el antecedente y se usa automáticamente (`initialfact`) como elemento condicional.
- Puede no haber ninguna acción en el consecuente, y la ejecución de la regla no tiene ninguna consecuencia.
- El antecedente es de tipo conjuntivo.
- Defrule Manager muestra la base de conocimiento.

3.3. Sintaxis del antecedente

El antecedente está compuesto de una serie de elementos condicionales (EC). Hay 8 tipos de elementos condicionales:

- EC Patrón

Consiste en un conjunto de restricciones, que se usan para verificar si se cumple un campo o slot de una entidad patrón. Este patrón posee las siguientes restricciones:

- Literales: Se define la restricción con un valor exacto de campo, es decir una constante, sin comodines ni variables.

`<restricción>::= <constante>`

Ejemplo:

```
; Aquí declaramos la base de hechos

(deffacts dato-hechos (dato 1.0 azul "rojo")
                      (dato 1 azul)
                      (dato 1 azul rojo))
```

```
(dato 1 azul ROJO)
(dato 1 azul rojo 6.9))
```

; Aquí definimos la regla

```
(defrule encontrar-datos (datos 1 azul rojo) =>)
```

**; Y finalmente ejecutamos el código que nos dará
; con que hecho se activará la regla**

```
CLIPS> (reset)
```

```
CLIPS> (facts)
```

```
f-0 (initial-fact)
```

```
f-1 (datos 1.0 azul "rojo")
```

```
f-2 (datos 1 azul)
```

```
f-3 (datos 1 azul rojo); Empareja con este
```

```
f-4 (datos 1 azul ROJO)
```

```
f-5 (datos 1 azul rojo 6.9)
```

For a total of 6 facts.

```
CLIPS> (agenda)
```

```
0 encontrar-datos: f-3
```

For a total of 1 activation

- Comodines

Indican que cualquier valor en esa posición de la entidad patrón es válido para emparejar con la regla. Hay dos tipos de comodines:

→ Monocampo: ?, empareja con un campo

→ Multicampo: \$?, empareja con 0 o más campos

```
<restricción>::= <constante> | ? | $?
```

Ejemplo:

```
; Declaramos la siguiente regla, la cual emparejaría con  
; los hechos f-3  
; y f-5 ya que el emparejamiento seria el siguiente:  
;  
; f-3 datos → datos  
;      ? → 1  
;      azul → azul
```

```

;      rojo → rojo
;      $? → [En este caso empareja con el conjunto vacío]
;
; f-5 datos → datos
;      ? → 1
;      azul → azul
;      rojo → rojo
;      $? → 6.9

```

```
CLIPS> (defrule encontrar-datos (datos ? azul rojo $?)=>)
```

```
CLIPS> (facts)
```

```
f-0 (initial-fact)
```

```
f-1 (datos 1.0 azul "rojo")
```

```
f-2 (datos 1 azul)
```

```
f-3 (datos 1 azul rojo)
```

```
f-4 (datos 1 azul ROJO)
```

```
f-5 (datos 1 azul rojo 6.9)
```

For a total of 6 facts.

```
CLIPS> (agenda)
```

```
0 encontrar-datos: f-5
```

```
0 encontrar-datos: f-3
```

For a total of 2 activations.

- Variables

Almacena el valor de un campo para después utilizarlo en otros elementos condicionales o consecuentes de la regla. Tenemos dos tipos:

→ Variable monocampo: ?<nombre-variable>

→ Variable multicampo: \$?<nombre-variable>

Y su estructura sería la siguiente:

```
<restricción> ::= <constante> | ? | $? | <variable-  
monocampo> | <variable-multicampo>
```

Cuando la variable aparece por 1ª vez, actúa como un comodín, pero el valor queda ligado al valor del campo, en caso de que vuelva a aparecer la variable, el valor debe coincidir con el valor asociado. Dicho requisito es funciona de manera local a la regla.

Ejemplo:

```
(defrule encontrar-datos-triples  
  (datos ?x ?y ?z)  
  =>  
  (printout t ?x " : " ?y " : " ?z crlf)  
)
```

CLIPS> (facts)

```
f-0      (initial-fact)  
f-1      (datos 2 azul verde)  
f-2      (datos 1 azul)  
f-3      (datos 1 azul rojo)
```

For a total of 4 facts. CLIPS> (run) 1 : azul : rojo 2 :
azul : verde

- Conectivas

Permite la unión entre conectivas y variables, este elemento utiliza los conectores lógicos and & (and), | (or) y ~ (not). Posee la siguiente estructura:

```
<restricción> ::= ? | $? | <restricción-conectiva>
```

```

<restricción-conectiva> ::= <restricción-simple> |
<restricción-simple> & <restricción-conectiva> |
<restricción-simple> | <restricción-conectiva>

<restricción-simple> ::= <término> | ~<término> <término>
::=<constante> | <variable-monocampo> | <variable-
multicampo>

```

El orden de precedencia es el siguiente: $\sim \rightarrow \& \rightarrow |$

Y las excepciones de este tipo de restricción es si la primera restricción es una variable seguida de la conectiva &, la primera restricción (la variable) se trata como restricción aparte

Ejemplo:

```
?x & rojo | azul == ?x & ( rojo | azul )
```

- De predicado

Se restringe un campo según el valor de verdad de una expresión lógica.

Se indica mediante dos puntos (:) seguidos de una llamada a una función predicado, la restricción se satisface si la función devuelve TRUE y normalmente se usan junto a una restricción conectiva y a una variable.

```

<término> ::= <constante> | <variable-monocampo>
                | <variable-multicampo>
                | :<llamada-a-función->

```

Ejemplo:


```
; Definimos una regla que compruebe si el valor de dato
; (datos 1)(datos 2)(datos rojo) es un número o no por lo
; por lo tanto asignamos a ?x el valor que lo acompaña
; y con el conector conector lógico & añadimos una
; condición de predicado para ver si es numero o no.
```

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

```
CLIPS> (assert (datos 1) (datos 2) (datos rojo))
```

```
<Fact-2>
```

```
CLIPS> (facts)
```

```
f-0 (datos 1)
```

```
f-1 (datos 2)
```

```
f-2 (datos rojo)
```

```
For a total of 3 facts.
```

```
CLIPS> (agenda)
```

```
0 predicado1: f-1
```

```
0 predicado1: f-0
```

```
For a total of 2 activations.
```

```
; Tras la ejecución del código podemos observar que se han
; activado f-0 y f-1 es decir (datos 1) y (datos 2)
```

CLIPS ofrece de manera nativa algunas funciones de predicado como puede ser floatp, integerp, numberp, stringp, symbolp, etc. Además de incorporar funciones de comparación

- De valor devuelto

Se utiliza el valor devuelto de una función para la restricción de un campo, el valor devuelto debe ser de uno de los tipos primitivos de datos y se sitúa en el patrón como si se tratase de una restricción literal en las comparaciones. Es indicado mediante el carácter

Su estructura es la siguiente:

```
<termino > ::= <constante> |  
              <variable-monocampo> |  
              <variable-multicampo> |  
              : <llamada-a-funcion>  
              = <llamada-a-funcion>
```

- Capturas de direcciones de hechos

A veces se desea realizar modificaciones, duplicaciones o eliminaciones de hechos en el consecuente de una regla, para ello es necesario que en la regla se obtenga el índice del hecho sobre el que se desea actuar.

```
<EC-patrón-asignado> ::=  
                        ?<nombre-variable> <- <EC-patrón>
```

· EC test

Comprueba el valor devuelto por una función, se satisface si el valor devuelto por la función no es FALSE y por lógica no se satisface si es FALSE.

Posee la siguiente estructura:

```
<EC-test> ::= (test <llamada-a-función>)
```

Ejemplo:

```
; Definimos una regla que compruebe si el valor absoluto  
; de la resta de dos variables es mayor o igual que tres
```

```

CLIPS> (defrule diferencia
      (dato ?x)
      (valor ?y)

      (test (>= (abs (- ?x ?y)) 3)) => )

CLIPS> (assert (dato 6) (valor 9))
<Fact-1> CLIPS> (facts)
f-0 (dato 6)
f-1 (valor 9)

For a total of 2 facts.
CLIPS> (agenda)
0      diferencia: f-0, f-1
For a total of 1 activation

```

· EC or

El elemento condicional or se satisface si los alguno de los elementos que lo compone, se cumple. Si se satisfacen varios ECs dentro del EC or, entonces la regla se disparará varias veces.

Estructura:

```
<EC-or> ::= (or <elemento-condicional>+)
```

Ejemplo:

```

; Definimos la regla para unas posibles tostadas, se
; activará siempre y cuando alguna de las dos reglas
; (tengo pan), (or (tengo mantequilla) (tengo aceite))

CLIPS> (defrule posibles-tostadas
      (tengo pan)
      (or (tengo mantequilla)

```

```

(tengo aceite))

=>

(assert (desayuno tostadas)))

CLIPS> (assert (tengo pan) (tengo mantequilla) (tengo
aceite))

; Una vez realizado el aserto, y al tener todos los
; elementos (pan, mantequilla y aceite), podemos ver
; los distintos pares de alimentos con los que podemos
; realizar las tostadas

<Fact-2> CLIPS> (agenda)

0 posibles-desayunos: f-0,f-2

0 posibles-desayunos: f-0,f-1

For a total of 2 activations

```

· EC and

El EC and se satisface si se satisfacen todos y cada uno de los elementos por los que está compuesto. Este tipo de elemento condicional permite mezclar and y or en el antecedente.

Estructura:

```

<EC-and> ::= (and <elemento-condicional>+)

```

Ejemplo:

```

; Ejemplo 2

; En este caso definimos una regla para realizar un
; posible desayuno. Será posible el desayuno en el caso
; de que tengamos zumo-natural o pan y aceite o leche y
; cereales

(defrule posibles-desayunos

(tengo zumo-natural)

```

```

(or (and (tengo pan)
          (tengo aceite))
    (and (tengo leche)
          (tengo cereales)))

=>

(assert desayuno tostadas)))

```

· EC not

El EC not se satisface si no se satisface el EC que contiene. Solo puede negarse un EC.

Estructura:

```
<EC-not> ::= (not <elemento-condicional>)
```

Ejemplo:

```

; Definimos la regla Homer-loco la cual se activa en el
; caso de que no haya ni tele ni cerveza que como
; consecuencia, Homer pierde la cabeza

(defrule Homer-loco
  (not (hay tele))
  (not (hay cerveza))

  =>

  (assert (Homer pierde la cabeza))

)

```

· EC exists

Permite comprobar si una serie de ECs se satisface por algún conjunto de hechos

Estructura:

```
<EC-exists> ::= (exists <elemento-condicional>+)
```

Ejemplo:

```
CLIPS> (defrule dia-salvado
  (objetivo salvar-el-dia)
  (heroe (estado desocupado))

=>

(printout t "El día está salvado" crlf))
CLIPS> (facts)

f-0 (initial-fact)
f-1 (objetivo salvar-el-dia)
f-2 (heroe (nombre spider-man) (estado desocupado))
f-3 (heroe (nombre daredevil) (estado desocupado))
f-4 (heroe (nombre iron-man) (estado desocupado))

For a total of 5 facts.

CLIPS> (agenda)

0 dia-salvado: f-1,f-4
0 dia-salvado: f-1,f-3
0 dia-salvado: f-1,f-2

For a total of 3 activations.
```

· EC forall

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

Estructura:

```
<EC-forall> ::= (forall <elemento condicional> <elemento-
condicional>+)
```

Ejemplo:

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

CLIPS> (reset)

CLIPS> (agenda)

0 todos-limpios: f-0,

For a total of 1 activation.

CLIPS> (assert (estudiante pepe) (lengua pepe)
(matematicas pepe)) <Fact-3>

CLIPS> (agenda)

CLIPS> (assert (historia pepe)) <Fact-4>

CLIPS> (agenda)

0 todos-limpios: f-0,

For a total of 1 activation.
```

- 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 de distintos de hechos, un hecho permanece mientras permanezca alguno de los que lo soportan logicamente. Todos los ECs incluidos en un logical van incluidos por un and implícito.

Estos pueden ser combinados con ECs and, or y not y solo los primeros ECs del antecedente pueden ser del tipo logical.

```
CLIPS> (defrule puedo-pasar
(logical (semaforo verde))
=>
(assert (puedo pasar)))
CLIPS> (assert (semaforo verde))
<Fact-0>
CLIPS> (run)
CLIPS> (facts)
f-0 (semaforo verde)
f-1 (puedo pasar)
For a total of 2 facts
```

3.4. Propiedades de una Regla

La declaración de propiedades se incluye tras el comentario y antes del antecedente, indicándola con la palabra declarada declare. Cada regla puede tener únicamente un declare.

Posee la siguiente estructura:

```
<declaración> ::= (declare <propiedad>) <propiedad> ::=
(salience <expresión-entera>)
```

3.4.1. Prioridad

Se indica en la definición de la regla con la palabra reservada salience, puede tomar valores entre -10000 y 10000, y su valor por defecto es 0 y su evaluación se hace de la siguiente manera:

- Cuando se define la regla (por defecto).
- Cuando se activa la regla.
- En cada ciclo de ejecución.

Cuando se activa la regla y en cada ciclo de ejecución entraría dentro de la llamada prioridad dinámica.

Ejemplo:

```
CLIPS> (clear)
CLIPS> (defrule primera

(declare (salience 10))

=>

(printout t "Me ejecuto la primera" crlf))

CLIPS> (defrule segunda

=>

(printout t "Me ejecuto la segunda" crlf))

CLIPS> (reset)
CLIPS> (run)

Me ejecuto la primera

Me ejecuto la segunda
```

3.5. Ejercicios propuestos

3.5.1. Ejercicio 0

```
; ej-00-reglas-basicas.clp

; =====

; Parte 1: Mostrar por pantalla una serie de alimentos

; Representación de la información mediante hechos ordenados
```

```

(deffacts hechos
  (comidas carne huevos pescado))

; Representación del conocimiento

(defrule r1
  (comidas $? ?x $?)
  =>
  (printout t ?x crlf)
)

; =====

; Parte 2: Valorar el uso en los elementos condicionales patrón,
; conectores lógicos

(deftemplate datos-B
  (slot valor)
)

(deffacts h1
  (datos-A verde) ;Hecho ordenado
  (datos-A azul) ;Hecho definido a partir de una plantilla
  (datos-B (valor rojo))
  (datos-B (valor azul)))

(defrule r1 (datos-A ~azul) =>)
(defrule r2 (datos-B (valor ~rojo&~verde)) =>)
(defrule r3 (datos-B (valor verde|rojo)) =>)

```

Este ejercicio consta de dos partes:

- En la primera parte tenemos elementos condicionales patrón que usan comodines.

Realizamos la llamada al constructor de hecho `deffacts hechos` y definimos un hecho llamado `comida`, que contiene tres elementos `carne`, `huevos` y `pescado`.

¿Acto seguido declaramos la regla con defrule compuesto por dos comodines multicampo y una variable monocampo (\$? ?x \$?) el emparejamiento se hace de la siguiente manera:

- \$? → carne
- ?x → huevos
- \$? → pescado

Y posteriormente se imprimen con printout.

· En la segunda parte tenemos definida una plantilla que posee un único campo (valor) posteriormente tenemos definida la base de hechos (h1) con cuatro hechos 2 definidos para hechos ordenados y 2 para hechos definidos a partir de plantilla.

Finalmente definimos 3 reglas (r1, r2 y r3) las cuales se activan con los siguientes hechos:

- r1: Se activará con cualquier hecho ordenado que posea un valor distinto de azul, por ejemplo, con el hecho (datos-A verde).
- r2: No se activa con ninguno ya que no hay ningún hecho definido mediante plantilla que no sea ni rojo ni verde.
- r3: Se activará con cualquiera de los dos hechos definidos mediante plantillas ya que da debe de ser verde o rojo.

3.5.2. Ejercicio 1

```
; ej-01-suma.clp

; Variables globales

(defglobal ?*s* = 0 )

; Hechos Iniciales

(deffacts vectores

  (v 1 2 3 4 5)
  (objetivo sumar)
```

```

)

(defrule sumar

(objetivo sumar)
(v $?antes ?x $?despues) ; Ejemplo de comodines multcampo

=>

(bind ?*s* (+ ?*s* ?x) ) ; Ligar valor a una variable global

)

(defrule presentaSuma

(declare (salience -10)) ;Priorizacion de las reglas

?ob<-(objetivo sumar)

=>

(printout t "Suma: " ?*s* crlf)
(retract ?ob)
(assert (suma ?*s*))
(bind ?*s* 0)

)

; *****

;Hechos iniciales

(deffacts vectores
(v 1 2 3 4 5)
(suma 0)

;(objetivo sumar)

)

(defrule r1

?f1 <- (v ?x $?despues)
?f2 <- (suma ?s)

=>

(retract ?f2)
(retract ?f1)
(assert (suma (+ ?s ?x)))
(assert (v ?despues))

)

```

Este ejercicio consta de dos partes:

- En la primera parte definimos una variable global (?*s*) igualada a 0
- Posteriormente declaramos un hecho llamado `vectores` (`v 1 2 3 4 5`), además de declaramos una regla que afecta al vector

3.5.3. Ejercicio 2

```
; ej-02-mayor.clp

; Mayor de un conjunto de números
; =====

(deffacts hechos-a
  (p 9) (p 1) (p 3)
  (resultado)
)

(defrule r1-mayor
  ?d<- (p ?x)
  (not (p ?y&:(> ?y ?x)))
=>
  (printout t ?x crlf)
  ;(retract ?d) ;Observar lo que ocurre al quitar el comentario
                ; y explicarlo
)

;Ordenar de mayor a menor

(deffacts hechos-a
  (p 9) (p 1) (p 3)
  (resultado)
)
```

```

(defrule r2-mayor-a-menor
  ?d<- (p ?x)
  (not (p ?y&:(> ?y ?x)))
  ?r <- (resultado $?resto)

=>

  (printout t ?x crlf)
  (retract ?r)
  (assert (resultado ?x $?resto))
  (retract ?d) ;Observar lo que ocurre al poner el comentario
)

; El mayor elemento de un vector
; =====

(deffacts hechos-b
  (v 28 99 1 4 7 9)
)

(defrule r2

  (v $?antes ?x $?despues )
  (not (v $?antes1 ?y&:(> ?y ?x) $?despues1) )

=>

  (printout t ?x crlf )
)

; Reordenar los elementos de un vector
; =====

(deffacts hechos-b
  (v 28 1 4 7 9)
  (nv)
)

(defrule r2

  ?f1<-(v $?antes ?x $?despues )
  ?f2<-(nv $?d)
  (not (v $?antes1 ?y&:(> ?y ?x) $?despues1) )

=>

  (printout t ?x crlf )
  (retract ?f1 ?f2)

```

```
(assert (v ?antes ?despues))
(assert (nv ?x ?d))

)
```

3.5.4. Ejercicio 3

```
; ej-03-suma-de-vectores.clp

; Definimos variables globales

(defglobal ?*s* = 0
           ?*p* = 1

)

(deffacts datos (vector 1 2 3 4))

(defrule suma ; suma todos los elementos de un vector

  ?ob <- (objetivo sumar)
          (vector $?antes ?y $?resto)
=>

  (bind ?*s* (+ ?*s* ?y))

)

(defrule presentaSuma
  (declare (salience -10))
  ?ob<-(objetivo sumar)
=>

  (printout t "Suma: " ?*s* crlf)
  (retract ?ob)
  (assert (suma ?*s*))
  (bind ?*s* 0)
  (printout t "Pulse c para continuar... " crlf)
  (read)

)

(defrule producto ; Multiplica todos los elementos de un
```

[illegible]


```

(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t " "   crlf)
(printout t "-----"   crlf)
(printout t "1. Suma: "   crlf)
(printout t "2. Producto: "   crlf)
(printout t "3. Obtener el mayor elemento: "   crlf)
(printout t "4. Salir: "   crlf)
(printout t "-----"   crlf)
)

(deffunction cont()

  (menu)
  (printout t "¿Elija una opcion?(1 2 3 4) ")
  (bind ?r (read))
  (while (neq ?r 1 2 3) do

    (printout t "¿Elija una opcion?(1 2 3 4) ")
    (bind ?r (read))

  )

  ;(printout t ?r crlf)

  (if (eq ?r 1)
    then
      (assert (objetivo sumar))
  )

  (if (eq ?r 2)
    then
      (assert (objetivo producto))
  )

  (if (eq ?r 3)
    then
      (assert (objetivo mayorElemento))
  )

  (if (eq ?r 4)
    then
      (printout t "Programa finalizado" crlf)
      (printout t "====="crlf)
      halt
  )

)

```

```
)
(defrule r
  (not (objetivo ?))
  =>
  (cont)
)
```

3.5.5. Ejercicio 4

```
; ej-04-ordenarCadenas.clp

(deffacts datos
  (nombre "aaz")
  (nombre "b")
  (nombre "c")
)

(defrule ordenar
  (nombre ?x)
  (not (nombre ?y:(eq (str-compare ?y ?x) 1)))
  =>
  (printout t ?x crlf)
)

(deftemplate persona
  (slot nombre)
  (slot apellidos)
  (slot dni)
  (slot ordenado)
)

(deffacts objetivos
  (objetivo presentar-personas)
)

; Representamos las personas

(deffacts personas
  (persona (nombre Juan)(apellidos Alvarez)(dni 1)(ordenado 0))
  (persona (nombre Pedro)(apellidos Martinez)(dni 2)(ordenado 0))
)
```

```

(persona (nombre Juan)(apellidos Fernandez)(dni 3)(ordenado 0))
(persona (nombre Pedro)(apellidos Valenzuela)(dni 4)(ordenado 0))
(persona (nombre Juan)(apellidos Sanchez)(dni 5)(ordenado 0))
(persona (nombre Pedro)(apellidos Zapata)(dni 6)(ordenado 0))
)

; Representacion del conocimiento
; =====

; Crear una regla para que presente los datos de las personas
; por pantalla ordenados alfabeticamente

(defrule presentacion

  (objetivo presentar-personas)

  ?p<- (persona (apellidos ?ap)(ordenado 0))

  (not (persona (apellidos ?y&:(eq (str-compare ?ap ?y) 1))
    (ordenado 0)))

  =>

  (printout t ?ap crlf)
  (modify ?p (ordenado 1))

)

```

3.5.6. Ejercicio 5

```

; ej-05.clp
;
; Representación de la información
; =====

; Definir una plantilla para representar relaciones de
; familiares de personas

; Primero definimos una plantilla para representar personas

(deftemplate persona

  (slot nombre)
  (slot apellidos)
  (slot dni)
  (slot ordenado)

)

```

```

; Definimos una plantilla para representar una relación familiar
(deftemplate rf
  (slot tipo)
  (slot dni-1)
  (slot dni-2)
)

(deffacts objetivos
  (objetivo presentar-personas)
)

; Representamos las personas

(deffacts personas
  (persona (nombre Juan)(apellidos Fernandez)(dni 1)(ordenado 0))
  (persona (nombre Pedro)(apellidos Martinez)(dni 2)(ordenado 0))
  (persona (nombre Juan)(apellidos Fernandez)(dni 3)(ordenado 0))
  (persona (nombre Pedro)(apellidos Valenzuela)(dni 4)(ordenado 0))
  (persona (nombre Juan)(apellidos Sanchez)(dni 5)(ordenado 0))
  (persona (nombre Pedro)(apellidos Zapata)(dni 6)(ordenado 0))
)

; Representamos las relaciones familiares

(deffacts relaciones
  (rf (tipo hijo_de) (dni-1 1) (dni-2 2) )
)

; Representación del conocimiento
; =====

; Crear una regla para que presente los datos de las personas
; por pantalla ordenados alfabeticamente

(defrule presentacion
  (objetivo presentar-personas)

  ?p<- (persona (apellidos ?ap)(ordenado 0))

  (not (persona (apellidos ?y&:(eq (str-compare ?y ?ap) 1))

  (ordenado 0)  ))

  =>

```

```

(printout t ?ap crlf)
(modify ?p (ordenado 1))
)

; Crear una regla para que presente los relaciones familiares
; de las personas por pantalla

(defrule presentacion-rf

  (rf (tipo ?t) (dni-1 ?dn1) (dni-2 ?dn2) )

  (persona (dni ?dn1) (nombre ?n1) (apellidos ?ap1) )

  (persona (dni ?dn2) (nombre ?n2) (apellidos ?ap2) )
  (test (neq ?dn1 ?dn2))

=>

  (printout t ?n1 " " ?ap1 "es " ?t " de " ?n2 " " ?ap2 crlf)
  (printout t ?dn1 " " ?dn2 crlf )

)

; INFERIR OTRA RELACIÓN FAMILIAR DE AQUI
; =====
; SI UNA PERSONA ES HIJA DE OTRA, ESTA OTRA ES PADRE DE LA
; PERSONA

```

3.5.7. Ejercicio 6

```

; ; ej-06.clp

; 2.- Programa para sumar y multiplicar los elementos de un
; vector

(defglobal ?*s* = 0
           ?*p* = 1

)

```

```

(deffacts datos (vector 1 2 3 4))

(defrule suma ;suma todos los elementos de un vector

  ?ob <- (objetivo sumar)
          (vector $?antes ?y $?resto)
=>

  (bind ?*s* (+ ?*s* ?y))
)

(defrule presentaSuma

  (declare (salience -10))
  ?ob<- (objetivo sumar)
=>

  (printout t "Suma: " ?*s* crlf)
  (retract ?ob)
  (assert (suma ?*s*))
  (bind ?*s* 0)
  (printout t "Pulse c para continuar... " crlf)
  (read)
)

(defrule producto ; Multiplica todos los elementos de un
                  ; vector haciendo uso de variables globales

  (objetivo producto)
  (vector $?antes ?y $?resto)
=>

  (bind ?*p* (* ?*p* ?y))
)

(defrule presentaProducto

  (declare (salience -10))
  ?ob<-(objetivo producto)
=>

  (printout t "Producto: " ?*p* crlf)
  (printout t "Pulse c para continuar... " crlf)
  (read)
  (retract ?ob)
  (assert (producto ?*p*))

```

```
(bind ?*p* 1)

)

; 3.
; a) Crear una función sin argumentos para presentar un menú
; de opciones,
; b) Crear una función cont () que seleccione una de las
; posibles opciones
; c)

(defun menu ()

  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t " "      crlf)
  (printout t "-----" crlf)
  (printout t "1. Suma: "   crlf)
  (printout t "2. Producto: " crlf)
  (printout t "3. Obtener el mayor elemento: " crlf)
  (printout t "4. Salir: "   crlf)
  (printout t "-----" crlf)

)

(defun cont()

  (menu)
  (printout t "¿Elija una opcion?(1 2 3 4) ")
  (bind ?r(read))
  (while (neg ?r 1 2 3) do
```

```

                (printout t "¿Elija una opcion?(1 2 3 4) ")
                (bind ?r (read))
            )
; (printout t ?r crlf)
    (if (eq ?r 1)
        then
            (assert (objetivo sumar))
        )

    (if (eq ?r 2)
        then
            (assert (objetivo producto))
        )

    (if (eq ?r 3)
        then
            (assert (objetivo mayorElemento))
        )

    (if (eq ?r 4)
        then

            (printout t "Programa finalizado" crlf)
            (printout t "====="crlf)

            halt
        )
    )
)

(defrule r
    (not (objetivo ?))
=>
    (cont)
)

```

3.5.8. Ejercicio 7

```

; ej-07-modulos.clp

(defmodule MAIN (export ?ALL))

(deftemplate MAIN::persona

```



```

(slot nombre)
(slot apellido)
(slot dni))

(deffacts datos

  (persona (nombre gabriel) (apellido garcia) (dni 1))
  (persona (nombre jose) (apellido garcia) (dni 2))
  (persona (nombre carlos) (apellido garcia) (dni 3))
  (persona (nombre juan) (apellido ruiz) (dni 4))
  (persona (nombre francisco) (apellido ruiz) (dni 5))
  (persona (nombre tomas) (apellido ruiz) (dni 6)))

(deftemplate MAIN::info

  (slot dni1)
  (slot dni2)
  (slot relacion))

(deffacts relaciones

  (info (dni1 1) (dni2 2) (relacion padre-hijo))
  (info (dni1 2) (dni2 3) (relacion padre-hijo))
  (info (dni1 4) (dni2 5) (relacion padre-hijo))
  (info (dni1 5) (dni2 6) (relacion padre-hijo)))

(defrule MAIN::foco

=>

  (focus A B C))

(defrule MAIN::fin
  (declare (salience -10))
  (objetivo fin)

=>

  (printout t "Fin del programa" crlf))

(defmodule A (import MAIN deftemplate info)

  (import MAIN deftemplate persona)
  (export ?ALL))

(defrule A::nietos

  (info (dni1 ?z) (dni2 ?x) (relacion padre-hijo))
  (info (dni1 ?x) (dni2 ?y) (relacion padre-hijo))

=>

  (assert (es-nieto dni1 ?y dni2 ?z)))

```

```

(defmodule B (export ?ALL)

  (import MAIN deftemplate persona)
  (import MAIN deftemplate info))

(defrule B::abuelos

  (info (dni1 ?z) (dni2 ?x) (relacion padre-hijo))
  (info (dni1 ?y) (dni2 ?z) (relacion padre-hijo))

=>

  (assert (es-abuelo dni1 ?y dni2 ?x)))

(defmodule C (export ?ALL)

  (import MAIN ?ALL)
  (import A ?ALL)
  (import B ?ALL))

(defrule C::mensaje
(declare (salience 10))

=>

(printout t "----Lista de nietos----" crlf))

(defrule C::lista

  (es-nieto dni1 ?x dni2 ?)
  (persona (nombre ?a) (apellido ?b) (dni ?x))

=>

  (printout t "Nombre: " ?a " " "Apellido: " ?b " " "DNI: " ?x crlf)
  (assert (objetivo fin))

)

```

4. Funciones definidas por el sistema

4.1. Funciones de predicado

4.1.1. Funciones de tipos

- (integerp <expresion>)

- Descripción

Comprueba si el argumento que recibe una función es un valor de tipo entero o no, retornando TRUE en caso afirmativo y FALSE en caso contrario

- Ejemplo

```
(deffunction esEntero (?numero)

  (if (integerp ?numero) then
    (printout t "El numero " ?numero " es entero" crlf)

    else

    (printout t "El numero " ?numero " no es entero" crlf)))
```

- Salida

```
CLIPS> (esEntero 5)
El numero 5 es entero
CLIPS> (esEntero 2.5)
El numero 2.5 no es entero
```

- (floatp <expresion>)

- Descripción

Comprueba que el argumento que recibe la función sea flotante

- Ejemplo

```
(deffunction esFlotante (?numero)
```

```
(if (floatp ?numero) then  
  (printout t "El numero " ?numero " es flotante" crlf)  
else  
  (printout t "El numero " ?numero " no es flotante" crlf)))
```

- Salida

```
CLIPS> (esFlotante 7.5)  
El numero 7.5 es flotante  
CLIPS> (esFlotante 7)  
El numero 7 no es flotante
```

- (numberp <expresión>)

-Descripción

Comprueba que el argumento pasado a la función sea un numero

- Ejemplo

```
(deffunction esNumero (?numero)  
  (if (numberp ?numero) then  
    (printout t "El numero " ?numero " es numero" crlf)  
  else  
    (printout t "El numero " ?numero " no es numero" crlf)))
```

- Salida

```
CLIPS> (esNumero "pepe")  
El numero pepe no es numero  
CLIPS> (esNumero 7)  
El numero 7 es numero
```

- (symbolp <expresión>)

-Descripción

Comprueba que el argumento pasado a la función sea de tipo symbol, es decir, verde, rojo, !!! (A diferencia del tipo string, este no va entre comillas).

- Ejemplo

```
(deffunction esSimbolo (?numero)
  (if (symbolp ?numero) then
    (printout t "El numero " ?numero " es simbolo" crlf)
  else
    (printout t "El numero " ?numero " no es símbolo" crlf)
  ))
```

- Salida

```
CLIPS> (esSimbolo verde)
El numero verde es simbolo
CLIPS> (esSimbolo "verde")
El numero verde no es simbolo
```

- (stringp <expresión>)

-Descripción

Comprueba que el argumento recibido por la función sea una cadena, podemos argumentar el caso anterior (diferencia entre simbolo y cadena)

- Ejemplo

```
(deffunction esCadena (?cadena)
  (if (stringp ?cadena) then
    (printout t "La variable " ?cadena " es una cadena" crlf)
  else
    (printout t "La variable " ?cadena " no es una cadena" crlf)
  ))
```

- Salida

```
CLIPS> (esCadena "Prueba")
La variable Prueba es una cadena
CLIPS> (esCadena Prueba)
La variable Prueba no es una cadena
```

· (lexemep <expresión>)

- Descripción

Comprueba que el argumento que recibe la función es una cadena o un símbolo.

- Ejemplo

```
(deffunction esTexto (?txt)
  (if (lexemep ?txt) then
    (if (stringp ?txt) then
      (printout t "La variable" ?txt "es una cadena" crlf)
    else
      (printout t ?txt " es un simbolo" crlf))
  else
    (printout t ?txt " no es ni cadena ni simbolo" crlf)))
```

- Salida

```
CLIPS> (esTexto pedro)
La variable pedro es un simbolo
CLIPS> (esTexto "Pedro")
La variable Pedro es una cadena
CLIPS> (esTexto 6)
La variable 6 no es ni cadena ni simbolo
```

· (evenp<expresión>)

- Descripción

Devuelve TRUE si es argumento que recibe la función es par y FALSE si este es impar

- Ejemplo

```
(deffunction esPar (?numero)

  (if (evenp ?numero) then
    (printout t "El numero " ?numero " es par" crlf)

    else

    (printout t "El numero " ?numero " es impar" crlf)))
```

- Salida

```
CLIPS> (esPar 5)
El numero 5 es impar
CLIPS> (esPar 6)
El numero 6 es par
```

- (oddp<expresión>)

- Descripción

Caso inverso de la función anterior en la que devuelve TRUE si el argumento que recibe la función impar y FALSE si es par.

- Ejemplo

```
(defunción esImpar (?numero)

  (if (oddp ?numero) then
    (printout t "El numero " ?numero " es impar" crlf)
  else
    (printout t "El numero " ?numero " es par" crlf)))
```

- Salida

```
CLIPS> (esImpar 3)
El numero 3 es impar
CLIPS> (esImpar 4)
El numero 4 es par
```

· (multifieldp <expresión>)

-Descripción

Devuelve TRUE si la variable es multicampo y FALSE si la variable no es multicampo

- Ejemplo

```
(multifieldp (create$ a 3 4.1))  
(multifieldp 1)
```

- Salida

```
CLIPS> (multifieldp (create$ a 3 4.1))  
TRUE  
CLIPS> (multifieldp (create$ ))  
TRUE  
CLIPS> (multifieldp 1)  
FALSE
```

4.1.2. Funciones aritméticas

· (eq <expresión1> <expresión2>+)

-Descripción

Compara el valor de la expresion1 con el valor de las expresiones restantes devuelve TRUE si coinciden las expresiones y FALSE si alguna de ellas no coincide.

- Ejemplo

```
(eq 4 4 4 4)  
(eq 4 4 4 1)
```


- Salida

```
CLIPS> (eq 4 4 4 1)
FALSE
CLIPS> (eq 4 4 4 4)
TRUE
```

· (neq <expresión1> <expresión2>+)

-Descripción

Caso inverso de la función anterior, en la que devuelve TRUE si no coinciden dichas expresiones y FALSE si coinciden

- Ejemplo

```
(neq 4 3 2 1)
(neq 4 4)
```

- Salida

```
CLIPS> (neq 4 3 2 1)
TRUE
CLIPS> (neq 4 4)
FALSE
```

(= <expresión-num1> <expresión-num2>+)

-Descripción

Compara 2 o más expresiones numéricas devolviendo TRUE si coinciden y FALSE si no lo son

- Ejemplo

```
(= 3 3.0)  
(= 3 3.0001)
```

- Salida

```
CLIPS> (= 3 3.0)  
TRUE  
  
CLIPS> (= 3 3.0001)  
FALSE
```

(<> <expresión-num1> <expresión-num2>+)

-Descripción

Comprueba que la expresión numérica 2 esté fuera del rango de la expresión numérica 1 devuelve TRUE si lo esta y FALSE si no lo esta

- Ejemplo

```
(<> 4 4.1)  
(<> 4 4.0)
```

- Salida

```
CLIPS> (<> 4 4.0)  
FALSE  
  
CLIPS> (<> 4 67)  
TRUE
```

(< <expresión-num1> <expresión-num2>+)

-Descripción

Comprueba que la expresión numérica 1 sea menor que la expresión numérica 2 devolviendo TRUE su la expresión numérica 2 es mayor que la 1 y FALSE si no lo es

- Ejemplo

```
( < 4 6 )  
( < 4 3 )
```

- Salida

```
CLIPS> ( < 4 6 )  
TRUE  
  
CLIPS> ( < 4 3 )  
FALSE
```

(<= <expresión-num1> <expresión-num2>+)

-Descripción

Comprueba que la expresión numérica 1 sea menor o igual que la expresión numérica 2 devolviendo TRUE si es mayor o igual la expresión numérica 2 que la 1 y FALSE si no lo es

- Ejemplo

```
( <= 4 6 )  
( <= 4 4 )  
( <= 4 3 )
```

- Salida

```
CLIPS> ( <= 4 6 )  
TRUE  
CLIPS> ( <= 4 4 )  
TRUE  
CLIPS> ( <= 4 3 )  
FALSE
```

(> <expresión-num1> <expresión-num2>+)

-Descripción

Comprueba que la expresión numérica 1 sea mayor que la expresión numérica 2 devolviendo TRUE si la expresión numérica 2 es menor que la 1 y FALSE si no lo es

- Ejemplo

```
(> 4 6)
(> 4 3)
```

- Salida

```
CLIPS> (> 4 6)
FALSE
CLIPS> (> 4 3)
TRUE
```

(>= <expresión-num1> <expresión-num2>+)

-Descripción

Comprueba que la expresión numérica 1 sea mayor o igual que la expresión numérica 2 devolviendo TRUE si es menor o igual la expresión numérica 2 que la 1 y FALSE si no lo es

- Ejemplo

```
(>= 4 6)
(>= 4 4)
(>= 4 3)
```

- Salida

```
CLIPS> (>= 4 6)
FALSE
CLIPS> (>= 4 4)
TRUE
CLIPS> (>= 4 3)
TRUE
```

4.1.3. Funciones lógicas

- (and<expresión>+)

-Descripción

Evalúa de manera lógica (mediante el producto lógico) las salidas de las funciones, es decir TRUE o FALSE

- Ejemplo

```
(and TRUE TRUE)
(and FALSE TRUE)
```

- Salida

```
CLIPS> (or TRUE TRUE)
TRUE
CLIPS> (or FALSE TRUE)
FALSE
```

- (or <expresión>+)

-Descripción

Evalúa de manera lógica (mediante la suma lógica) las salidas de las funciones, es decir TRUE o FALSE, también se podría comprobar directamente introduciendo lo siguiente:

- Ejemplo

```
(or FALSE FALSE)
(or FALSE TRUE)
```

- Salida

```
CLIPS> (or FALSE FALSE)
FALSE
CLIPS> (or FALSE TRUE)
TRUE
```

- (not <expresión>)

-Descripción

Actúa como complemento de salidas es decir niega la salida de las funciones

- Ejemplo

```
(not TRUE)
(not FALSE)
```

- Salida

```
CLIPS> (not TRUE)
FALSE
CLIPS> (not FALSE)
TRUE
```

4.2 Funciones multicampo

- (create\$)

-Descripción

El valor de retorno de la función create\$ es el valor del multicampo, independientemente del valor o tipo de argumentos.

- Ejemplo

```
(create$ rojo 1 verde "pan")  
(create$)
```

- Salida

```
CLIPS> (create$ rojo 1 verde "pan")  
(rojo 1 verde "pan") ; Devuelve los campos  
  
CLIPS> (create$)  
( ) ; Devuelve el campo vacio
```

- (length\$ <expr-multicampo>)

-Descripción

Retorna la cantidad de argumentos que posee el multicampo creado

- Ejemplo

```
(length$ (create$ pan 4 verde "rojo"))
```

- Salida

```
CLIPS> (length$ (create$ pan 4 verde "rojo"))  
4
```

- (nth\$ <expr-entera> <expr-multicampo>)

-Descripción

Devuelve el valor de argumento multicampo que especifica la expresión numérica entera

- Ejemplo

```
(nth$ 3 (create$ pan 3 siete "Verde"))
```

- Salida

```
CLIPS> (nth$ 3 (create$ pan 3 siete "Verde"))  
siete
```

- (member\$ <expresión> <expr-multicampo>)

-Descripción

Retorna la posición de la primera ocurrencia de una determinada expresión en el multicampo, en caso de que no haya ocurrencias, devuelve FALSE

- Ejemplo

```
(member$ platano (create$ rojo "verde" 1 platano))  
  
(member$ 1 (create$ rojo "verde" 1 platano 1)  
(member$ azul (create$ rojo "verde" 1 platano))  
(member$ (create$ 1 2)(create$ 1 2 4 5))
```


- Salida

```
CLIPS> (member$ platano (create$ rojo "verde" 1
platano))
4
CLIPS> (member$ 1 (create$ rojo "verde" 1 platano 1))
3
CLIPS> (member$ azul (create$ rojo "verde" 1 platano))
FALSE
CLIPS> (member$ (create$ 1 2)(create$ 1 2 4 5))
(1 2)
```

- (subsetp <expr-multicampo> <expr-multicampo>)

- Descripción

Devuelve TRUE si los elementos del primer campo multivaluado estan en el segundo campo multivaluado independientemente del orden, y FALSE en caso contrario

- Ejemplo

```
(subsetp (create$ verde 1) (create$ verde "rojo" 1 2
azul))
(subsetp (create$ 2 3) (create$ verde "rojo" 1 2 azul))
```

- Salida

```
CLIPS> (subsetp (create$ verde 1) (create$ verde "rojo"
1 2 azul))
TRUE
CLIPS> (subsetp (create$ 2 3) (create$ verde "rojo" 1 2
azul))
FALSE
```

· **(subseq\$<expr-multicampo> <exp-entera-inicio> <exp-entera-fin>)**

- Descripción

Funciona de la misma manera que la función anterior solamente que en esta función sí importa el orden

- Ejemplo

```
(subseq$ (create$ 1 2 3 4 5 6 7 8 9) 4 7)
```

- Salida

```
CLIPS> (subseq$ (create$ 1 2 3 4 5 6 7 8 9) 4 7)
(4 5 6 7)
```

· **(first\$ <expr-multicampo>)**

- Descripción

Devuelve el primer valor del campo multivaluado

- Ejemplo

```
(first$ (create$ 1 2 3 4 5 5 6))
```

- Salida

```
CLIPS> (first$ (create$ verde rojo azul amarillo
negro))
(verde)
```

- **(rest\$ <expr-multicampo>)**

- **Descripción**

Es la función complementaria a first\$, retorna todos valores restantes menos el primero

- **Ejemplo**

```
(rest$ (create$ 1 2 3 4 5 5 6))
```

- **Salida**

```
CLIPS> (rest$ (create$ 1 2 3 4 5 5 6))  
(2 3 4 5 5 6)
```

- **(explode\$ <expr-cadena>);**

- **Descripción**

Convierte todo elemento de una cadena a valores de un multicampo, los simbolos, floats, enteros o nombres de variables son convertidos a cadenas

- **Ejemplo**

```
(explode$ "Esto es \"una\" pru)eba")
```

- **Salida**

```
CLIPS> (explode$ "Esto es \"una\" pru)eba")  
(Esto es "una" pru ") eba)
```

- **(implode\$ <expr-multicampo>);**

- **Descripción**

Convierte todos los valores de un multicampo en cadena

- Ejemplo

```
(implode$ (create$ Esto es una prueba))
```

- Salida

```
CLIPS> (implode$ (create$ Esto es una prueba))  
"Esto es una prueba"
```

- (insert\$ <expr-multicampo> <exp-enteraN> <expresión>+);

- Descripción

Inserta una expresión en la posición indicada del multicampo, en caso de que haya algún valor en dicha posición lo desplaza a la posición n+1

- Ejemplo

```
(insert$ (create$ verde 1 "pan") 3 huevo)
```

- Salida

```
CLIPS> (insert$ (create$ verde 1 "pan" ) 3 huevo)  
(verde 1 huevo "pan")
```

**- (replace\$ <expr-multicampo> <exp-entera-inicio> <exp-entera-fin>
<expresión>+)**

- Descripción

Reemplaza la expresión indicada en la posición indicada del multicampo(o rango de posiciones)

- Ejemplo

```
(replace$ (create$ verde 1 "pan") 1 1 cambio)
(replace$ (create$ verde 1 "pan") 1 2 cambio)
```

- Salida

```
CLIPS> (replace$ (create$ verde 1 "pan") 1 1 cambio)
(cambio 1 "pan")
CLIPS> (replace$ (create$ verde 1 "pan") 1 2 cambio)
(cambio "pan")
```

- (delete\$ <expr-multicampo> <exp-entera-inicio> <exp-entera-fin>)

- Descripción

Elimina elementos de un multicampo especificados previamente mediante un rango

- Ejemplo

```
(delete$ (create$ azul 1 pan "verde" coche vaca) 2 3)
(delete$ (create$ azul 1 pan "verde" coche vaca) 2 4)
```

- Salida

```
CLIPS> (delete$ (create$ azul 1 pan "verde" coche vaca)
2 3)
(azul "verde" coche vaca)
CLIPS> (delete$ (create$ azul 1 pan "verde" coche vaca)
2 4)
(azul coche vaca)
```

4.3. Funciones de cadena

· (str-cat<expresión>*)

- Descripción

Esta función concatena una 1 o más cadenas, símbolos, etc. Retornando una cadena resultado de la concatenación del resto.

- Ejemplo

```
(str-cat "Esto es " "una " "Pr" "ue""ba")  
(str-cat "Esto es " "una " "Pr" 1 "ba")
```

- Salida

```
CLIPS> (str-cat "Esto es " "una " "Pr" "ue""ba")  
"Esto es una Prueba"  
CLIPS> (str-cat "Esto es " "una " "Pr" 1 "ba")  
"Esto es una Pr1ba"
```

· (sym-cat<expresión>*)

- Descripción

Realiza una concatenación de 1 o más cadenas, símbolos, etc. retornando un dato de tipo simbolo.

- Ejemplo

```
(sym-cat<expresión>*)
```

- Salida

```
CLIPS> (sym-cat "Esto es " 1 "Pr" "ue" "ba")  
Esto es una Prueba  
  
CLIPS> (sym-cat "Esto es " 1 "Pr" "ue" "ba")  
Esto es 1Prueba
```

· (str-length<expr-cadena-o-símbolo>)

- Descripción

Retorna la longitud de un elemento de cadena o simbolo

- Ejemplo

```
(str-length "Cual es la longitud de esta cadena?")  
(str-length Longitud)
```

- Salida

```
CLIPS> (str-length "Cual es la longitud de esta  
cadena?")  
36  
  
CLIPS> (str-length Longitud)  
9
```

· (sub-string<expr-entera> <expr-entera> <expr-cadena>)

-Descripción

Retorna una subcadena comprendida entre los extremos que se especifican

- Ejemplo

```
(sub-string 3 7 "Esto es una cadena de prueba?")
```

- Salida

```
CLIPS> (sub-string 3 7 "Esto es una cadena de prueba?")  
"to es"
```

· **(str-compare<expr-cadena-o-símbolo> <expr-cadena-o-símbolo>);**

-Descripción

Compara letra a letra las dos cadenas, devolviendo 0 si son iguales, un valor menor que 1 si la primera es más corta que la segunda y un valor mayor que 1 si la primera es mayor que la segunda

- Ejemplo

```
(str-compare "prueba" "prueba")  
(str-compare "prueba" "pruebaaaa")  
(str-compare "pruebaaaa" "prueba")
```

- Salida

```
CLIPS> (str-compare "prueba" "prueba")  
0  
CLIPS> (str-compare "prueba" "pruebaaaa")  
-1  
CLIPS> (str-compare "pruebaaaa" "prueba")  
1
```

4.4. Funciones de E/S

· **(open<nombre-fichero> <nombre-lógico> <modo>);**

-Descripción

Devuelve TRUE si el fichero en cuestión, se ha abierto correctamente y FALSE en caso contrario (por ejemplo que no exista el archivo). Si se trabaja en Windows hay que cambiar "\" por "/" a la hora de especificar la ruta

- Ejemplo

```
(open "nombreFichero.txt" file "r")  
  
(open "C:/Users/David/GoogleDrive/Universidad/1516/Sistemas  
Inteligentes/Practicas/Sesion 9_0505/EjemplosSIN0505.clp"  
file "r")
```


- Salida

```
CLIPS> (open "nombreFichero.txt" file "r")
FALSE
CLIPS> (open "C:/Users/David/Google Drive/Universidad/1516/
SistemasInteligentes/Practicas/Sesion9_0505/EjemplosSIN0505.c
lp" file "r")
TRUE
```

- (close[<nombre-lógico>]);

-Descripción

Función que recibe el nombre del "puntero" del fichero que se ha abierto previamente y cuya función es cerrarlo devolviendo TRUE si se ha cerrado correctamente.

- Ejemplo

```
(close file)
```

- Salida

```
CLIPS> (open "C:/Users/David/GoogleDrive/Universidad/1516
/Sistemas Inteligentes/Practicas/Sesion9_0505/
EjemplosSIN0505.clp" file "r")
TRUE

CLIPS> (close file)
TRUE
```

- (printout <nombre-lógico> <expresión>*);

-Descripción

Escribe en el fichero la cadena pasada por parámetro, se debe abrir el fichero en cuestión en modo escritura o adición.

- Ejemplo

```
(open "C:/Users/David/Google Drive/Universidad/1516/Sistemas
Inteligentes/Practicas/Sesion 9_0505/ejemploFichero.clp" file
"a")

(printout file "Prueba N") ; ...

(close file)
```

- Salida

```
CLIPS> (open "C:/Users/David/Google
Drive/Universidad/1516/Sistemas Inteligentes/Practicas/Sesion
9_0505/ejemploFichero.clp" file "a")
TRUE
CLIPS> (printout file "Prueba 1" crlf)
CLIPS> (printout file "Prueba 2" crlf)
CLIPS> (printout file "Prueba 3" crlf)
CLIPS> (printout file "Prueba 4" crlf)
CLIPS> (close file)
TRUE
```

- (read[nombre-lógico>]);

-Descripción

Esta función lee del fichero abierto en modo lectura, cada llamada a la función lee hasta el siguiente salto de línea / espacio, devuelve los elementos por los cuales está formado el texto y en caso de que llegue al final del fichero, devolvería EOF.

- Ejemplo

```
(read file)
```

- Salida

```
CLIPS> (open "C:/Users/David/GoogleDrive/Universidad/1516
/Sistemas Inteligentes/Practicas/Sesion9_0505/
ejemploFichero.clp" file "r")
```

```
TRUE
CLIPS> (read file)
Prueba1
CLIPS> (read file)
Prueba2
CLIPS> (read file)
Prueba3
CLIPS> (read file)
Prueba4
CLIPS> (read file)
EOF
CLIPS> (close file)
TRUE
```

· **(readline[nombre-lógico]);**

-Descripción

Dicha funcion devuelve la línea completa incluyendo espacios, para ello abrimos el fichero en modo lectura y ejecutamos el comando readline hasta que la salida sea EOF

- Ejemplo

```
(readline file)
```

- Salida

```
CLIPS> (readline file)
EOF

CLIPS> (close file)
TRUE
```

4.5. Funciones matemáticas

· **(+ <expr-num> <expr-num>+);**

-Descripción

Suma dos o más números

- Ejemplo

```
(+ 3 2)
```

- Salida

```
CLIPS> (+ 3 2)  
5
```

- (- <expr-num> <expr-num>+)

-Descripción

Resta dos o más números

- Ejemplo

```
(- 3 2)
```

- Salida

```
CLIPS> (- 3 2)  
1
```

- (* <expr-num> <expr-num>+);

-Descripción

Multiplica dos o más números

- Ejemplo

```
(* 3 2)
```

- Salida

```
CLIPS> (* 3 2)
6
```

- (/ <expr-num> <expr-num>+)

-Descripción

Divide dos o más números (contempla la parte decimal)

- Ejemplo

```
(/ 3 2)
```

- Salida

```
CLIPS> (/ 3 2)
1.5
```

- (div <expr-num> <expr-num>+)

-Descripción

Divide 2 o más números (devuelve la parte entera)

- Ejemplo

```
(div 3 2)
```

- Salida

```
CLIPS> (div 3 2)
1
```

· (mod<expr-num> <expr-num>)

- Descripción

Realiza la operación módulo de dos números

- Ejemplo

```
(mod 3 2)
```

- Salida

```
CLIPS> (mod 3 2)  
1
```

· (sqrt<expr-num>)

- Descripción

Realiza la raíz cuadrada de un numero

- Ejemplo

```
(sqrt 3)
```

- Salida

```
CLIPS> (sqrt 3)  
1.73205080756888
```

· (** <expr-num> <expr-num>);

- Descripción

Realiza la operación exponente, hay que indicar base (expresión numérica 1) y exponente (expresión numérica 2)

- Ejemplo

```
(** 3 2)
```

- Salida

```
CLIPS> (** 3 2)
9.0
```

- (round<expr-num>);

-Descripción

Redondea dependiendo de la parte decimal, si la parte decimal esta entre [0.0 , 0.5] redondea a la baja en el caso contrario, es decir, si la parte decimal es mayor que 0.5 se redondea a la alza (0.5 , 0.9]

- Ejemplo

```
(round (sqrt 3))
```

- Salida

```
CLIPS> (round (sqrt 3)); El valor de sqrt(3) es 1.7320508...
2
```

- (abs<expr-num>);

- Descripción

Devuelve el valor absoluto del numero pasado a la funcion

- Ejemplo

```
(abs -3)
```

- Salida

```
CLIPS> (abs -3)
3
```

· (max<expr-num>+)

- Descripción

Devuelve el valor máximo de un conjunto de valores

- Ejemplo

```
(max 3 4 6 1 6 6 2 10)
```

- Salida

```
CLIPS> (max 3 4 6 1 6 6 2 10)
10
```

· (min<expr-num>+);

- Descripción

Devuelve el valor mínimo de un conjunto de valores

- Ejemplo

```
(min 3 4 6 1 6 6 2 10)
```

- Salida

```
CLIPS> (min 3 4 6 1 6 6 2 10)
1
```

4.6. Funciones procedurales

· (bind<variable> <expresión>*);

-Descripción

Declara una variable y se le asigna uno o varios valores

- Ejemplo

```
(bind ?variable "Esto es una prueba")

(bind ?variable2 "Esto es una prueba" "Añadimos otra
cadena mas")
```

- Salida

```
CLIPS> (bind ?variable "Esto es una prueba")
"Esto es una prueba"
CLIPS> (printout t ?variable crlf)
Esto es una prueba

CLIPS> (bind ?variable2 "Esto es una prueba" "Añadimos
otra
cadena mas")
("Esto es una prueba" "Añadimos otra cadena mas")
CLIPS> (printout t ?variable2 crlf)
("Esto es una prueba" "Añadimos otra cadena mas")
```

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

-Descripción

Estructura condicional if-else

- Ejemplo

```
(deffunction esImpar (?numero)

(if (oddp ?numero) then

    (printout t "El numero " ?numero " es impar" crlf)

else

    (printout t "El numero " ?numero " es par" crlf)

))
```

- Salida

```
CLIPS> (esImpar 7)
El numero 7 es impar
CLIPS> (esImpar 8)
El numero 8 es par
```

- (switch <expr-prueba> <sentencia-caso> [<sentencia-defecto>])

-Descripción

Estructura condicional *switch-case*. Otra manera de expresar dicha estructura es:

<sentencia-caso> ::= (case <expr-comparación>then<acción>*)

<sentencia-defecto> ::= (default<acción>*)

- Ejemplo

```
(deffunction foo (?val)
  (switch ?val
    (case 1 then (printout t "Este es el caso 1"
      crlf))
    (case 2 then (printout t "Este es el caso 2"
      crlf))
    (default (printout t "Este caso no esta
      contemplado" crlf)))
  )
```

- Salida

```
CLIPS> Loading Selection...
Defining deffunction: foo
CLIPS> (foo 1)
Este es el caso 1
CLIPS> (foo 2)
Este es el caso 2
CLIPS> (foo 3)
Este caso no esta contemplado
```

- (while<expresión> [do] <acción>*)
(loop-for-count<rango> [do] <acción>*)

<rango> ::=
<índice-final> |
(<variable> <índice-final>) |
(<variable> <índice-inicio> <índice-final>)
<índice-inicio> ::= <expr-entera>
<índice-final> ::= <expr-entera>
(return[<expresión>]) (break);

-Descripción

Bucle iterativo do-while. Una manera simplificada de esta estructura es:

```
(while (< condicion >)
    ... código ...
)
```

- Ejemplo

```
(deffunction contadorSimple (?i)
    (while (> ?i 0)
        (printout t ?i crlf)
        (bind ?i (- ?i 1)))
)
```

- Salida

```
CLIPS> Loading Selection...
Defining deffunction: contadorSimple
CLIPS> (contadorSimple 5)
5
4
3
2
1
FALSE
```

5. Bibliografía

- Diapositivas Tema 1 de la asignatura:

http://moodle.uco.es/m1516/pluginfile.php/183484/mod_resource/content/0/Practicas/tema1.pdf

- Elementos básicos de CLIPS:

<http://www.uco.es/users/sventura/misc/TutorialCLIPS/TutorCLIPS02.htm>

- Diapositivas Tema 6 de la asignatura:

http://moodle.uco.es/m1516/pluginfile.php/183489/mod_resource/content/0/Practicas/tema6.pdf

- Basic Programming Guide CLIPS:

<http://clipsrules.sourceforge.net/documentation/v630/bpg.pdf>

- Diapositivas Tema 11 de la asignatura:

http://moodle.uco.es/m1516/pluginfile.php/183496/mod_resource/content/0/Practicas/tema11.pdf