Inteligencia Artificial

CLIPS

ROBOTIKA
& SISTEMA
AUTONOMOEN
IKERKETA TALDEA

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Qué es CLIPS?

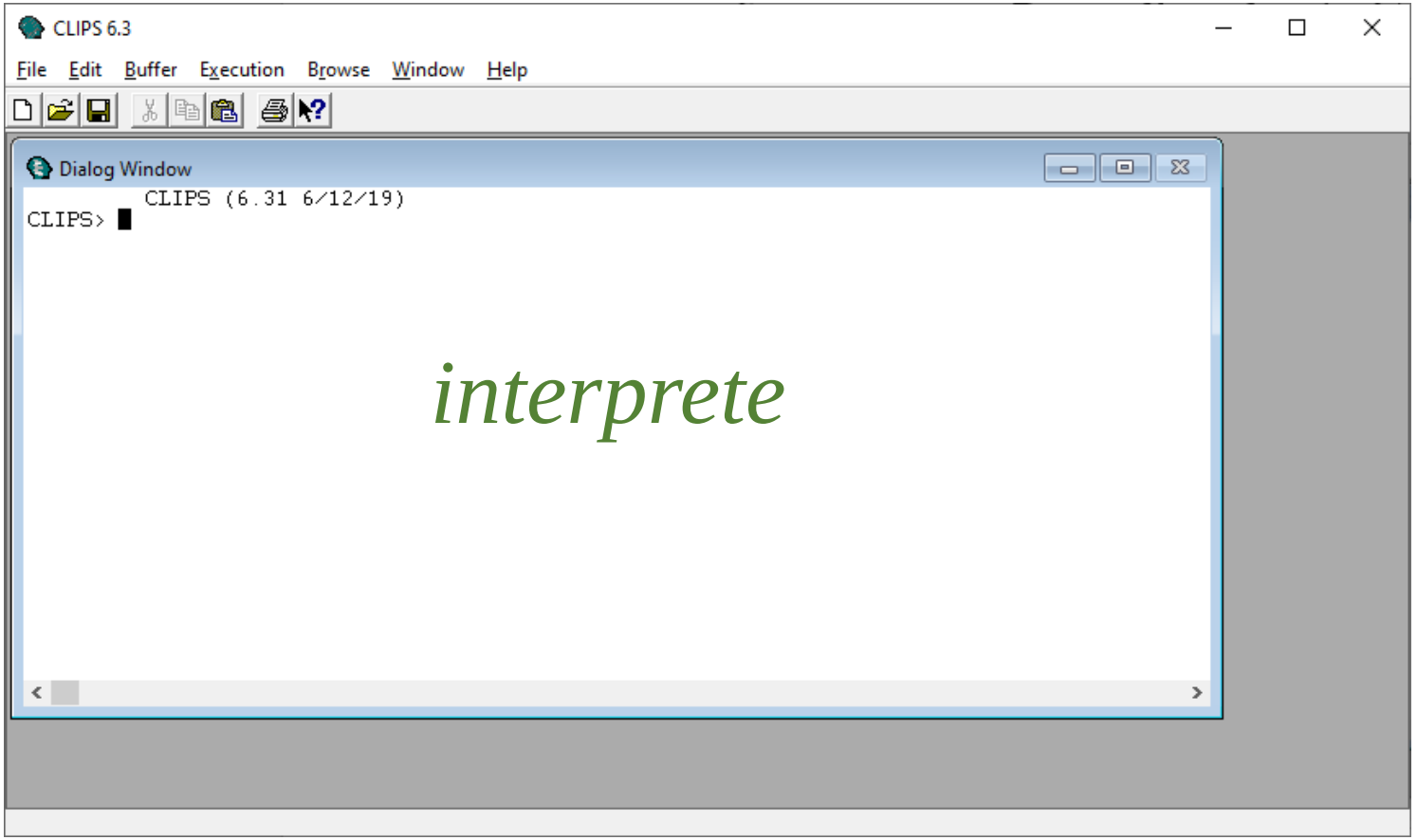
- CLIPS (C Language Integrated Production System) es una **herramienta para el desarrollo de sistemas expertos** (SE) creada por la Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center.
- Los orígenes de CLIPS se remontan a **1984**.
- Se diseñó para **tratar de modelar el conocimiento humano**.
- CLIPS **permite integración completa con otros lenguajes de programación** como C, Java, .Net, Ada, etc.
- CLIPS es un **entorno completo para la construcción de SE basados en reglas** y/o objetos.

https://sourceforge.net/projects/clipsrules/files/CLIPS/

- Versión que **no** hay que instalar: clips_windows_executables_630.zip
- Versión que hay que instalar: clips_631_windows_64_bit_installer.msi

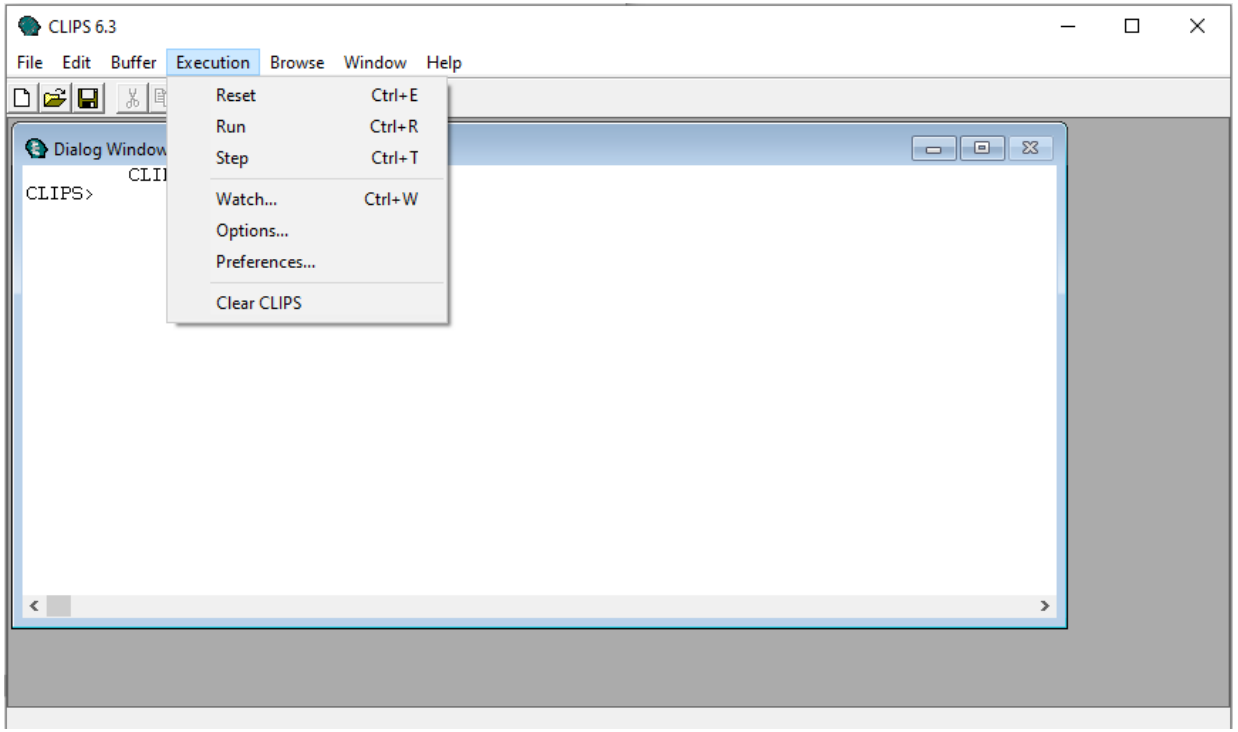
CLIPS

- Disponible para varias plataformas. La versión para Windows dispone interface y línea de comando. Haciendo clic sobre el icono de esta última (CLIPSIDEx) lanzarás la ventana que acompaña a este texto. Para salir teclea ***(exit)*** o acude a la barra de menú **File → exit**.



CLIPS

- ***(exit)*** Cierra la interfaz CLIPS.
- ***(run)*** Lanza la ejecución del programa CLIPS actualmente cargado. Se puede indicar el número de secuencias de ejecución; (*run 3), (run 40)* (**solo cuando se usan reglas**).
- ***(clear)*** Elimina todo de la memoria (variables, hechos y reglas), equivalente a cerrar y reabrir CLIPS.
- ***(reset)*** Elimina los valores de los hechos y variables de la memoria (las reglas no), volviendo a default.
- ***(load)*** Carga un fichero.
- ***(watch <elemento>)*** Permite realizar depuración del programa.



Programación básica en CLIPS

- CLIPS **proporciona tres elementos básicos** para escribir programas:
  - **Tipos primitivos de datos**: **para representar información**.
  - **Constructoras**: estructuras sintácticas identificadas por una **palabra reservada del lenguaje que permiten definir funciones, reglas, hechos, clases**, etc., que alteran el entorno de CLIPS añadiéndolas a la base de conocimiento. Los constructores no devuelven ningún valor. Su sintaxis es similar a la definición de funciones.
  - **Funciones** propias: para **manipular los datos**. Devuelven un valor.

Elementos básicos: Tipos de datos

- **No hay que definir el tipo!**
- Reales (***float***)**:** 1.5, -0.7, 3.5e-10
- Enteros (***integer***): 1, -1, +3, 65
- Símbolos (***symbols***): Cualquier secuencia de caracteres **que no empiece con número**. **Distingue entre mayúsculas y minúsculas**. Ej.: Febrero, febrero, fuego, B35, fiebre
- Cadenas (***strings***): **"**Deben estar entre comillas**"**.
- Dentro de los paréntesis el primer elemento **SIEMPRE** es una función

```
CLIPS> 1.5
CLIPS> (integer 1.3) → 1     (integer 1.9) → 1        casting
CLIPS> (integerp 1.3) → FALSE                         pregunta tipo dato
CLIPS> hola
CLIPS>"hola"
```

Elementos básicos: Constructores

- Palabras reservadas **(sentencias):**
  - De momento usaremos:
    - **deffunction**: Para definir funciones
    - **defglobal**: Para definir variables globales
  - Más adelante (**con reglas**):
    - **deftemplate**: Para definir plantillas
    - **deffacts**: Para definir hechos
    - **defrule**: Para definir reglas

Variables

- Con la función ***bind*** se asigna un valor (**equivalente a =**).
- **Variables simples:** Para guardar solamente un único valor, usaremos ***?nombre-var***

  (bind ?var1 5) → ?var1 = 5

- **Variables multicampo:** para guardar más de un valor **(“lista”)**, usaremos ***$?nombre-var***
  - *el comando **create$** sirve para crear un multicampo, es decir, una “lista”*

    (bind $?var2 5 hola 89 “adios”) → $?var 2 = (5 hola 89 “adios”)

    (bind $?var3 (create$ 5 hola 89 “adios”)) → $?var 3 = (5 hola 89 “adios”)

- **Variables globales:** deben nombrarse delimitadas por ***** y emplearemos el constructor ***defglobal***

  (defglobal ?*var1* = 7) ***(defglobal ?*var2* = (create$ adios 5 “hola"))***

- ***(reset)***: las variables globales retoman su valor original (**de cuando fueron creadas**).
  - Se puede ver el valor de las variables globales en “Window → Globals windows”.

- ***(clear)*** se elimina todo de la memoria (equivalente a cerrar CLIPS).

Entrada y salida

- **Entrada:** La función ***(read)*** permite a CLIPS **leer información proporcionada por el usuario**. El programa se detiene, a la espera de que el usuario teclee el dato.

(bind ?var1 (read))

  - La función ***read*** lee **solo hasta el primer espacio en blanco o salto de línea**.
  - Si queremos introducir una serie de elementos, separados por espacios, debemos utilizar ***(readline)***. La función ***(readline)*** retorna una cadena **(String).**

(bind ?var1 (readline))

- Para convertir un String a un conjunto de símbolos usaremos la función ***explode$***

(bind ?var1 (explode$ (readline)))

Si se teclea ***hola mundo 100 adiós*** → (hola mundo 100 adiós) → **multicampo**

Entrada y salida

- **Salida:** El comando ***printout*** permite imprimir por pantalla como por fichero:

  Pantalla → ***(printout t "¡Hola!" crlf)*** *Fichero* → ***(printout File1 "¡Hola!" crlf)***

  crlf: salto de línea **(no es necesario)**

- **Por fichero:** Antes de escribir en el fichero, hay que abrirlo (*modo escritura **w**, lectura **r** y apéndice **a***), indicando correctamente la ruta.

  (open "ruta/miFichero.txt" File1 "w") → abre fichero en modo escritura

  (close File1) → cierra el fichero

  (printout File1 "hola mundo" crlf) → escribe datos en el fichero y salta de línea.

  (format File1/nil "Nombre: %s y Edad: %d" "Ekaitz" 18) → escribe datos en File1 (fichero) o nil (pantalla). **No necesita crlf**.

  (bind ?palabra (read File1) → lee del fichero sólo hasta el primer espacio en blanco o EOF.
  (bind ?palabra (readline File1) → lee del fichero hasta salto de línea o EOF.

  (rename "miFichero.txt" "tuFichero.txt") → cambia el nombre del fichero
  (remove "tuFichero.txt") → elimina el fichero

Estructuras de control

- **Sentencia condicional:**

(if** (<condición>) **then

(<acción>)

***[else** (<acción>)])*

```
(if (= ?a ?b) then
      (printout t “son iguales” crlf)
 else
      (printout t “no son iguales" crlf)
)
```

- **Sentencia repetitiva:**

(loop-for-count (**<var> <inicio><final>**)** [do] <acción>**)
(while (**<condición>) [do] (<acción>**))
(foreach** <var> <varMulti> (<accion>**))

***NOTA: los elementos entre corchetes son opcionales y no hace falta poner* [ ]**

```
(loop-for-count 2
      (printout t "Hola mundo" crlf)
)

(loop-for-count (?i 0 2)
      (loop-for-count (?j 1 3)
            (printout t ?i " " ?j crlf)
       )
)
```

```
(foreach ?a (create$ 1 2 3)
      (printout t ?a )
)        → 1 2 3

(bind ?v 4)
(while (> ?v 0)
      (printout t "v es " ?v crlf)
      (bind ?v (- ?v 1))
)
```

Ejemplo funciones multicampo

- Ejemplo ***progn***: similar a *foreach*
  - permite realizar un conjunto de acciones sobre cada campo de un valor multicampo**.**

	Salida
(progn$ (?var (create$ abc def ghi)) (printout t "-->" ?var "<--" crlf))	--> **abc** <-- --> **def** <-- --> **ghi** <--

Estructuras de control

- **Switch:**

(switch *<expresión-test>*
(case *<comparación>* ***then*** *<acción>**)***
...
(case *<comparación>* ***then*** *<acción>**)***
[(default *<acción>**)])***

  - **Ejemplo:**

```
(bind ?vari 2)
    (switch ?vari
        (case 1 then
            (printout t "variable = 1" crlf))
        (case 2 then
            (printout t "variable = 2" crlf))
     )
```

Operadores lógicos y matemáticos

- **Operadores lógicos:** and, or, not
  - **Ejemplo:**

```
(bind ?vari1 2)
(bind ?vari2 20)
(if (and(= ?vari1 2)(= ?vari2 20)) then
      (printout t "Hola" crlf)
else
      (printout t "Adios" crlf)
)
```

- **Matemáticos:** abs, div, float, integer, max, min, +, -, *, /
  - (abs <número>) → devuelve el valor absoluto del argumento
  - (div <número> <número> +) → devuelve la división **(entero)** **(div 6 3 2)=1**
  - (float <número>) → devuelve el argumento convertido a float (Casting)
  - (integer <número>) → devuelve el argumento convertido a integer (Casting)
  - (max | min <número> <número> +) → devuelve el argumento máximo o mínimo
  - (<op> <número> <número>+)
    - Donde **op:** **+** (suma todos los elementos), ***** (producto de todos los elementos)
    - **-** , / (resta o divide el primer elemento con los demás (**/ devuelve float**))

Operadores lógicos y matemáticos

- **Más funciones matemáticos :** exp, log, mod, sqrt, **, random
  - (exp <número>) → devuelve ***e*** elevado a la potencia que indica el argumento
  - (log <número>) → devuelve el logaritmo en base ***e*** del argumento
  - (log10 <número>) → devuelve el logaritmo **en base 10** del argumento
  - (mod <número> <número>) → devuelve el resto de la división de los argumentos
  - (pi) → devuelve el valor del número π
  - (round <número>) → devuelve el valor del argumento al entero más cercano
  - (sqrt <número>) → devuelve la raíz cuadrada del argumento
  - (** <número> <número>) → devuelve el valor del primer argumento elevado a la potencia del segundo argumento
  - (random [incio][fin]) → devuelve un número entero aleatorio dentro de ese rango

- **Trigonométricas:** (<*f*> <*expresión-numérica*>)
  - Donde ***f*** → **sin | cos | tan | cot | sec | tanh | sinh | ...**

Strings / Símbolos

- **Funciones con Strings y/o Símbolos:**
  - (lowcase | upcase <String|Símbolo>) → convierte a minúsculas | mayúsculas el símbolo o String

    CLIPS> (lowcase "HOLA") → "hola"
  - (str-cat <String|Int|Símbolo>*) → concatena todos los parámetros y **devuelve un String**

    CLIPS> (str-cat hola "HOLA" 2) → "holaHOLA2"
  - (sym-cat <String|Int|Símbolo>*) → concatena todos los parámetros y **devuelve un Símbolo**

    CLIPS> (sym-cat funciones "Strings" 1) → funcionesStrings1
  - (str-compare <String|Símbolo>) → devuelve el resultado de comparación:
    - 0 si ambos parámetros son idénticos.
    - 1 si el 1er string (o símbolo) > 2o string (o símbolo)
    - -1 si el 1er string (o símbolo) < 2o string (o símbolo)

    CLIPS> (str-compare hola "Hola") → 1 (porque minúscula > mayúscula → h > H)

    CLIPS> (str-compare atring hola) → -1 (porque f > a)
  - (str-index <String1> <String2>) → devuelve el índice en la que se encuentra Str1 en Str2 (puede usarse con símbolos)

    CLIPS> (str-index "mu" "holamundo") → 5

    CLIPS>(str-index "mu" "hola") → False

Strings / Símbolos

- **Más funciones con Strings:**
  - (str-length <String|Símbolo>) → devuelve la longitud

    CLIPS> (str-length “holamundo”) → 9

  - (sub-string <inicio><fin><String|Sím>) → devuelve una subcadena **(String)** entre inicio-fin

    CLIPS> (sub-string 2 4 “holamundo”) → “ola”

    CLIPS> (sub-string 4 2 “holamundo”) → “”

  - (eval <String>) → trata de convertir en función el String

    CLIPS> (eval “(+ 3 4)”) → 7

  - (string-to-field <String|Símbolo>) → convierte una cadena en un campo

    CLIPS> (string-to-field “3.4”) → 3.4

Elementos multicampo

- **Manipulación de valores multicampo:** todas devuelven lo creado o modificado
  - (create$ <expresión>*) → crea un valor multicampo

    CLIPS> (create$ (+ 3 4) xyz "texto" (/ 8 4)) → (7 xyz "texto" 2.0)

  - (delete$ <expresiónMultiC><inicio><fin>) → borra los campos en ese rango

    CLIPS> (delete$ (create$ a b c d e) 3 4) → (a b e)

  - (explode$ <String>) → crea un valor multicampo a partir de los campos del String

    CLIPS> (explode$ "1 2 abc \" el coche\"") → (1 2 abc "el coche")

    - Para diferenciar que hay un string dentro de otro string ponemos la \

  - (first$ <expresiónMultiC>) → devuelve el primer campo en un multicampo

    CLIPS> (first$ (create$ a b c d e)) → (a)

  - (implode$ <expresiónMultiC>) → devuelve un String formado de los campos

    CLIPS> (implode$ (create$ a b c d e)) → "a b c d e"

  - (insert$ <expresiónMultiC><pos><expresiónMulti|Simple>) → inserta en posición

    CLIPS> (insert$ (create$ a b c d) 1 x) → (x a b c d)

Elementos multicampo

- (length$ <expresiónMultiC>) → devuelve la cantidad de campos

CLIPS> (lengtht$ (create$ a b c d)) → 4

- (member$ <expresiónSimple><expresiónMultiC>) → devuelve la posición

CLIPS> (member$ z (create$ a b c d)) → false **CLIPS> (member$ c (create$ a b c d)) → 3**

- (nth$ <pos><expresiónMultiC>) → devuelve el campo de esa posición

CLIPS> (nth$ 3 (create$ a b c d)) → c **CLIPS> (nth$ 9 (create$ 1 2 3 4)) → nil**

- (replace$ <expresiónMultiC><inicio><fin><reemplazaMultiC|Simple>) → remplaza en ese rango

CLIPS> (replace$ (create$ a b c) 2 3 x) → (a x) **CLIPS> (replace$ (create$ a b c) 3 3 x) → (a b x)**

- (rest$ <expresiónMultiC>) → devuelve un valor multicampo quitando el primero

CLIPS> (rest$ (create$ a b c)) → (b c)

- (subseq$ <expresiónMultiC><inicio><fin>) → devuelve un valor multicampo de ese rango

CLIPS> (subseq$ (create$ a b c d) 3 4) → (c d)

- (subset <expresiónMultiC><enEstaexpresiónMultiC>) → devuelve si el primer multicampo es parte del otro

CLIPS> (subsetp (create$ 1 2)(create$ 1 2 3)) → TRUE

Funciones definidas por usuario

- Se utiliza el constructor ***deffunction*:**

(deffunction <nombre> [<comentario>] (<parámetro>* [<parámetro-comodín>])

<acción>*)

```
(deffunction suma (?a ?b)
            (bind ?suma (+ ?a ?b))
)
```

- Las funciones devuelven la última expresión. Se puede usar ***(return expresión)***

```
(deffunction suma (?a ?b)
            (bind ?suma (+ ?a ?b))
            (return ?suma)
)
```

Funciones definidas por usuario

- ***list-deffunctions*** permite listar las funciones definidas
- Otro ejemplo:

```
(deffunction mostrar-params (?a ?b $?c)
        (printout t ?a " " ?b " and " (length ?c) " extras: " ?c crlf)
)

> (mostrar-params 1 2) → 1 2 and 0 extras: ()
> (mostrar-params a b c d) → a b and 2 extras: (c d)
```

- **NOTA: solamente admite un multicampo y siempre ha de ser el último parámetro**
- Como pasar mas de una variable multicampo?
  - En la función definimos una variable simple **?a** pero en la llamada le pasamos una variable multicampo

```
(deffunction trampa (?a $?b)
                …
)
```

Funciones definidas por usuario

- En vez de usar el interprete de Clips, es recomendable crear nuestros programas con un editor, con extensión **.clp**

```
(defglobal ?*a* = 3)
(deffunction suma (?b)
                (bind ?suma (+ ?*a* ?b))
)
…
```

⇦ fichero**.clp**

- Desde la interfaz de Clips, cargamos el fichero donde tenemos nuestras funciones y realizamos las llamadas