

Introducción a Jess

- ❑ Instalación
- ❑ El lenguaje Jess: símbolos, valores, variables, listas, funciones
- ❑ Tratamiento de hechos
 - A) Hechos ordenados
 - B) Hechos no ordenados: deftemplate
 - C) Gestión de la memoria de trabajo
- ❑ Reglas. Patrones y acciones
- ❑ Operaciones de Entrada y Salida interactivas
- ❑ Utilidades auxiliares. Instrucciones de control.
- ❑ Estrategias de resolución de conflictos
- ❑ Estructura básica de un programa en Jess
- ❑ Materiales de referencia

Instalación

- ❑ Desempaqueta el Jess71p2.zip en el directorio que quieras
 - ❑ mejor donde tienes instalados el resto de tus programas
- ❑ Utiliza un editor de textos que compruebe paréntesis
 - ❑ El más simple: *notepad++*
 - ❑ Si conoces *emacs* es cómodo
- ❑ Ejecución en línea de comandos
 - ❑ **cd "c:\Program Files\Jess71p2\bin"**
 - ❑ **jess.bat**
 - ❑ Aparece el prompt: **Jess>**

El lenguaje Jess

- ❑ **Jess** es un lenguaje principalmente declarativo que permite **construir sistemas basados en reglas**
- ❑ Su sintaxis es tipo Lisp
- ❑ Cualquier componente de Jess se expresa en forma de lista de símbolos
- ❑ Entre símbolos puede haber cualquier número de espacios en blanco
- ❑ Se distingue entre mayúsculas y minúsculas

Símbolos

- ❑ Pueden incluir:

- ❑ a-z A-Z 0-9 \$ * . = + / < > _ ? #

- ❑ No pueden empezar por:

- ❑ 0-9

- ❑ \$? & = caracteres reservados para usos especiales

- ❑ Símbolos especiales:

- ❑ nil, TRUE, FALSE, crlf

Valores

- ❑ Los valores en Jess pueden ser:
 - ❑ Símbolos
 - ❑ Juan amarillo respuesta22 _ejemplo
 - ❑ Números
 - ❑ 56 47.8 5654L 6.0E4
 - ❑ Cadenas
 - ❑ “esto es un ejemplo”
 - ❑ Listas de símbolos, números, cadenas
 - ❑ (a b c d) (+ 3 5) (“Pregunta 1 “ “Nombre”) () (eq (3 5))
- ❑ Comentarios
 - ❑ Líneas que empiezan por ;
 - ❑ Si el comentario abarca varias líneas /* comentario */

Variables

- ❑ Cadenas que empiezan por ?
 - ❑ ?pregunta ?nombre ?edad
- ❑ Las variables no son tipadas aunque los valores lo sean
- ❑ Para asignar un valor a una variable se usa la función bind
 - ❑ (bind ?edad 18)
- ❑ Variables multivaluadas: empiezan por \$?
 - ❑ \$?apellido
- ❑ Variables globales. Se definen con la función
 - ❑ (defglobal ?*variable* = valor-por-defecto)
 - ❑ (defglobal ?*x* = 7)

Listas

- ❑ Una lista es una secuencia ordenada de valores
- ❑ Creación de listas: función `create$`
 - ❑ `(bind ?ejemplolista (create$ a b c d e))`
- ❑ Manejo de listas:
 - ❑ `(nth$)` devuelve el enésimo elemento
 - ❑ `(first$)` devuelve el primer elemento
 - ❑ `(rest$)` devuelve la lista sin su primer elemento

Funciones

- ❑ Para definir funciones:
 - ❑ (deffunction <nombre> (<parametro>*) [<comentario>] <expresión>*)
- ❑ Ejemplos de funciones predefinidas:
 - ❑ (facts) lista todos los hechos presentes en la memoria de trabajo
 - ❑ (rules) lista todas las reglas que haya en la base de reglas
 - ❑ (watch) activa los mecanismos de depuración en Jess
- ❑ Para ver todas las funciones predefinidas:
 - ❑ <http://www.jessrules.com/jess/docs/71/functions.html>
- ❑ Otras construcciones Jess (deffacts, defrule, etc.):
 - ❑ <http://www.jessrules.com/jess/docs/71/constructs.html>

A) – Hechos ordenados

- ❑ Secuencia de literales separados por espacios
 - ❑ Codifican la información según la posición
 - ❑ El primer literal suele representar una relación entre los restantes
 - ❑ Los restantes son como atributos o slots sin nombre

(convenio)
(alumnos Juan Luis Pedro)
(lista-de-la-compra pan leche arroz)
- ❑ Para incluirlos en la Base de Hechos se asertan (no se declaran)

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

```
(assert (temperatura 25) )
```
- ❑ El encaje o *matching* con el LHS de una regla
 - ❑ Los literales deben estar en el mismo orden que en la regla
- ❑ Se usan para conceptos con poca información
 - ❑ Es difícil manejar sus atributos por separado
 - ❑ Para ese manejo se usan los hechos NO ordenados

B) - Hechos no ordenados: deftemplate

- ❑ Define un tipo de hecho, tiene varios slots (atributos) con nombre
(deftemplate persona

```
(slot nombre (type SYMBOL) (default "sin nombre"))  
(multislot apellidos (type SYMBOL))  
(slot edad (type NUMBER)(default (+ 10 15)))  
(slot estado (type SYMBOL)(allowed-values soltero  
libre casado viudo)(default soltero)))
```

- ❑ Para cada slot se puede definir:
 - ❑ el tipo: type (valores posibles -> siguiente transparencia)
 - ❑ Valor por defecto: default (admite cualquier operación)
 - ❑ Valores permitidos: allowed-values
 - ❑ Slot multivaluados: multislot
- ❑ Se necesita establecer cada hecho con **assert** o **deffacts** (se ve después)

Sintaxis completa deftemplate

```
((deftemplate template-name
  [extends template-name]
  ["Documentation comment"]
  [(declare (slot-specific TRUE | FALSE)
    (backchain-reactive TRUE | FALSE)
    (from-class class name)
    (include-variables TRUE | FALSE)
    (ordered TRUE | FALSE))]
  (slot | multislot slot-name                                /*-- multislot es multivaluado --*/
    [(type ANY | INTEGER | FLOAT | NUMBER | SYMBOL | STRING |
      LEXEME | OBJECT | LONG)]
    [(default default value)]
    [(default-dynamic expression)]
    [(allowed-values expression+)]*))
```

C) - Gestión de la Memoria de Trabajo (MT)

- ❑ (deffacts ...) define un conjunto de hechos iniciales que se cargan en la MT al hacer (reset)
- ❑ (assert <hecho>) añade hecho a la MT
- ❑ (retract <índice-hecho>) elimina hecho de la MT
- ❑ (facts) lista los hechos existentes en la MT
- ❑ (clear) elimina todos los hechos de la MT
- ❑ (reset)
 - ❑ elimina todos los hechos de la MT y las activaciones de la agenda
 - ❑ añade initial-fact y los hechos definidos con deffacts
 - ❑ añade las variables globales con su valor inicial
 - ❑ selecciona el módulo main

Crear hechos iniciales con deffacts

□ Sintaxis

```
(deffacts deffacts-name ["Documentation comment"] fact* )
```

□ Ejemplo

```
(deffacts alumnos "mi clase"      /* -- hechos iniciales --*/  
  (persona (nombre Pepe)(apellidos Gomez Garcia))  
  (persona (nombre Juan)(edad 25)))
```

→ No olvidar hacer:

```
(reset) /*- borra todos los hechos de MT, añade los deffact */  
(facts) /*---- lista los hechos actuales en la M.T. --*/  
f-0    (MAIN::initial-fact)  
f-1    (MAIN::persona (nombre Pepe) (edad 25) (estado soltero) (apellidos Gomez  
Garcia))  
f-2    (MAIN::persona (nombre Juan) (edad 25) (estado soltero) (apellidos ))
```

□ Si quiero volver a ejecutar deffacts, debo ejecutar reset de nuevo

Etiquetas temporales

- ❑ Son índices relativos al orden de creación de hechos
- ❑ $f - 0$ es el initial-fact, creado automáticamente por Jess
- ❑ Al resto de hechos se les van asignando índices sucesivos:
 - ❑ $f - 1$
 - ❑ $f - 2, \dots$
- ❑ Identifican de forma única cada hecho
- ❑ Cuando se elimina un hecho, nunca se reasigna el índice a otro hecho
- ❑ Cuando se modifica un hecho se mantiene el mismo índice

Reglas

❑ Sintaxis:

```
(defrule <nombre-regla>           ;; para eliminar : undefrule
[<documentación opcional>]       ;; Se pone entre " "
[(declare (salience <num>))]     ;; prioridad de ejecución
(patrón 1)
(patrón 2)
...
(patrón N)
=>
(acción 1)
(acción 2)
...
(acción M)
)
```

- ❑ Ver el contenido de una regla (ppdefrule calcular-precio)
- ❑ Una regla sin LHS se ejecuta solo cada vez que se ejecute el **reset**.

Parte izquierda de las reglas: Patrones

- ❑ La parte izquierda de las reglas suele incluir patrones:
 - ❑ Variables (`?edad`)
 - ❑ Variables anónimas (comodines, no importa su valor)
 - ❑ `?` Se equipara con un valor
 - ❑ `$?` Se equipara con múltiples valores
 - ❑ Expresiones con variables y conectivas lógicas
 - ❑ `not` (`~`), `and` (`&`), `or` (`|`)
 - ❑ Test de expresiones lógicas (`test (< ?x 18)`)
 - ❑ Condiciones complejas precedidas de :
 - ❑ (`persona (edad ?x&: (> ?x 18))`)
- ❑ Las condiciones de una regla están implícitamente conectadas con `and`. Si necesitamos un `or` entonces hay que dividir la regla en dos.

Parte derecha de las reglas: Acciones

- ❑ Son acciones implícitamente conectadas con **and**

- ❑ Tipos de acciones:
 - ❑ Crear un hecho (**assert**)
 - ❑ Eliminar un hecho (**retract**)
 - ❑ Modificar un hecho (**modify**)
 - ❑ Llamar a una función
 - ❑ Asignar un valor a una variable (**bind**)
 - ❑ Entrada / Salida (**printout**, **read**, **readline**)
 - ❑ Parar la ejecución (**halt**)

Reglas: Ejemplo

❑ Ejemplo:

```
(deffacts ini
  (letra 1 c)
  (letra 2 a)
  (letra 3 c)
  (letra 4 b)
  (letra 5 a)
  (letra 6 b) )

((defrule r2  "para ordenar letras, como en ejemplo clase"
  ?h1<-(letra ?i c) ;;asigna el hecho (letra 1 c) a la variable ?h1, ?i = 1
  ?h2<-(letra ?j a)
  (test (eq ?i (- ?j 1)))      ;; se ejecuta regla si ?i = ?j - 1
=>
  (retract ?h1)                ;; quita el hecho (letra 1 c) de la MT
  (retract ?h2)
  (assert (letra ?i a)) ;; incluye en la MT el hecho (letra 1 a)
  (assert (letra ?j c))))
```

Modificar desde fuera la ejecución de la regla

- ❑ Permite usar una regla con diferentes fines :

- ❑ busco: que cumplan cierto estado y tengan menos de 30 años

- ❑ Al repetir `?est` fuerza que coincida su valor en los hechos que equiparen

```
(deftemplate busca (slot estado)) ;; un hecho para indicar qué busco
```

```
(defrule busca-candidato
```

```
  (busca (estado ?est)) ;; persona y busca han de tener el mismo ?est
```

```
  ?candidato <- (persona (nombre ?nom) (edad ?ed) (estado ?est))
```

```
                ;; asigno un hecho a la variable ?candidato
```

```
  (test (< ?ed 30)) ;; solo ejecuto regla si cumple test
```

```
=>
```

```
  (modify ?candidato (estado libre)) ;; cambia el estado de candidato
```

```
  (assert (tengo candidato))
```

```
  (printout t "mi candidato es: " ?nom " estado anterior: " ?est  
                                                     crlf) )
```

```
(reset)
```

```
(assert (busca (estado soltero))) ;; decido buscar solteros
```

```
(run)
```

Operaciones de Entrada y Salida interactivas - I

- ❑ Permite introducir un hecho entre comillas

```
(defrule inserta-hecho    ; no tiene LHS: se dispara si (reset) + (run)
    =>                    ; para escribir un texto al disparar regla
    (printout t "Escribe un hecho como cadena" crlf)
    (assert-string (read))) ; para leer un hecho
(reset)
(run)
Escribe un hecho como cadena    ; el ordenador escribe esto
"(persona (nombre NEO) )"      ; el usuario escribe eso
                                ; el sistema añade el hecho:
;;    (persona (nombre NEO) (apellidos ) (edad 25) (estado soltero))
```

Operaciones de Entrada y Salida interactivas – I I

- ❑ Permite introducir una línea y construir un hecho concatenando paréntesis

```
(defrule lee-línea
```

```
=>
```

```
(printout t "Introduce datos." crlf)
```

```
(bind ?cadena (readline))
```

```
(assert-string (str-cat "(" ?cadena "))))
```

```
Introduce datos                                ; el ordenador escribe esto
```

```
persona                                         ; el usuario escribe eso
```

```
                                                ; el sistema añade el hecho:
```

```
; (persona (nombre "sin nombre") (apellidos ) (edad 25) (estado  
soltero))
```

- ❑ Ejecución condicional : para poner apellidos a quien no tenga

```
(defrule poner-apellidos ;; solo equiparan personas sin apellidos
```

```
?persona <-(persona (nombre ?nombre) (apellidos))
```

```
=>
```

```
(printout t crlf "Introduce apellidos para " ?nombre ": " )
```

```
(modify ?persona (apellidos (read))))
```

Otras utilidades auxiliares

Listar las templates definidas (nativas y definidas por el usuario)

```
(list-deftemplates)
```

Trazar lo que va pasando (watch , unwatch)

```
(watch facts)(watch rules)(watch activations)(watch all)
```

Comprobar el tipo de una expresión o valor

```
(numberp <exp>) (stringp <exp>) (integerp <exp>)
```

Definir módulos para organizar las reglas

```
(defmodule <nombre>)
```

Cambiar de módulo

```
(focus <módulo>+) (declare (auto-focus TRUE)) (return)
```

Instrucciones de control (representación no declarativa)

- ❑ (if <exp> then <accion>* [elif <exp> then <accion>*]*
[else <accion>*])
- ❑ (while <exp> [do] <accion>*)
- ❑ (foreach <var> <lista> <accion>*)
- ❑ (declare salience <num>)
 - ❑ definiendo esta propiedad dentro de una regla, se establece su prioridad. Cuanto mayor sea el número mayor será la prioridad de la regla.

Estrategias de resolución de conflictos

- ❑ El motor de inferencia de Jess disparará las reglas aplicables por orden decreciente de prioridad (**salience**).
- ❑ Si hay varias reglas aplicables con la misma prioridad (o no se han definido prioridades) Jess utilizará **por defecto** la estrategia LIFO (**depth**): disparar antes las reglas activadas más recientemente.
- ❑ La estrategia FIFO (**breadth**) dispara las reglas de igual prioridad en el orden en que han sido activadas.
- ❑ (**set-strategy <estrategia>**) permite seleccionar la estrategia que utilizará el intérprete de Jess (**depth** o **breadth**)

Estructura básica de un programa en Jess

```
; definición de plantillas  
(deftemplate ...)  
  
...  
  
; definición de hechos iniciales  
(deffacts ...)  
  
...  
  
; definición de reglas  
(defrule ...)  
  
...  
  
(reset)  
  
(run)
```

Materiales de referencia

- Friedman-Hill, Ernest

Jess in action [Recurso electrónico] : rule-based systems in Java / Ernest Friedman-Hill Publicación Greenwich, Conn. : Manning, c2003

http://cisne.sim.ucm.es/record=b2548266~S6*spi

- Strauss, Martin

Jess. The Java Expert System Shell. Abril 2007
(tutorial disponible en Campus Virtual)

- Ejemplo de sistema con módulos

- taxadvisor.clp

- Jess 7.1 Manual

<http://www.jessrules.com/jess/docs/71/>

- Ejemplos disponibles en Jess71p2/examples/jess