JESS

Jess es un acrónimo para **Java Expert System Shell**. Es un motor de reglas y scripts para la plataforma Java, es decir, escrito en un ambiente enteramente del lenguaje Java por Ernest Friedman-Hill en el laboratorio Nacional de Sandia en Livermore, Canadá en 1995. Jess es un shell para la construcción de sistemas expertos y un lenguaje de scripts escrito totalmente en lenguaje java de Sun Microsystems. Puede usarse de dos formas: Como una máquina de reglas, y como un lenguaje de programación de propósito general.
Puede usarse en aplicaciones de línea de comando, aplicaciones gráficas, servlets y applets. La máquina de inferencia de Jess usa una forma mejorada del algoritmo RETE para asociar las reglas con la base de conocimiento.

Nota: Los Scripts son puros escritos solo en el lenguaje Jess sin código java.
Hay tres formas para representar conocimiento en Jess:

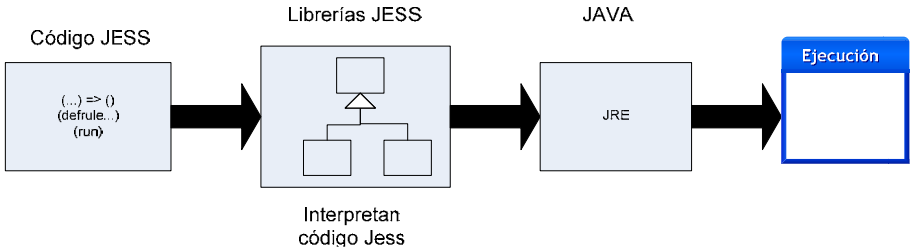
- **Reglas**, las cuales están primordialmente dirigidos al conocimiento heurístico basado en la experiencia.
- **Funciones**, las cuales están primordialmente dirigidos al conocimiento procedural.
- **Programación orientada a objeto**, también primordialmente pretendido para el conocimiento procedural.

CARACTERISTICAS:

- Es un motor de reglas para plataformas JAVA.
- Provee una interfaz que realiza una programación basada en reglas para el desarrollo de sistemas expertos.
- Puede utilizar todas las capacidades de JAVA.
- Jess (Java Expert System Shell) es una librería para programadores, escrita en Java, que permite implementar sistemas expertos mediante la utilización de un lenguaje sencillo
- JESS es una librería escrita en Java.
- Antes de usar la librería JESS se debería tener instalado la maquina virtual de Java.

    - El componente JRE es el que realmente ejecuta los programas.

    Las librerías JESS sirven como un intérprete para el Lenguaje JESS.



- JESS es una especialización de LISP.

  ¿Qué es LISP?

  - Lenguaje de programación.
  - El elemento fundamental de LISP son las listas.
  - Aplicaciones:
    - Sistema de álgebra computacional **Máxima**.
    - **Emacs** o **GNU Emacs** es un editor de texto altamente extensible y configurable.
    - **ACL2** es un demostrador automatizado de teoremas.

- Software y documentación.

  - El software y la documentación puede ser adquirida en la página http://herzberg.ca.sandia.gov/jess/

- Aplicaciones:

  - (EMINUS) es un Sistema Experto basado en Web que brinde soporte técnico en línea a los usuarios del Sistema de Educación Distribuida de la Universidad Veracruz (México). De esta manera, el servicio estará disponible las 24 horas del día, brindando atención a los problemas oportunamente y con calidad técnica.

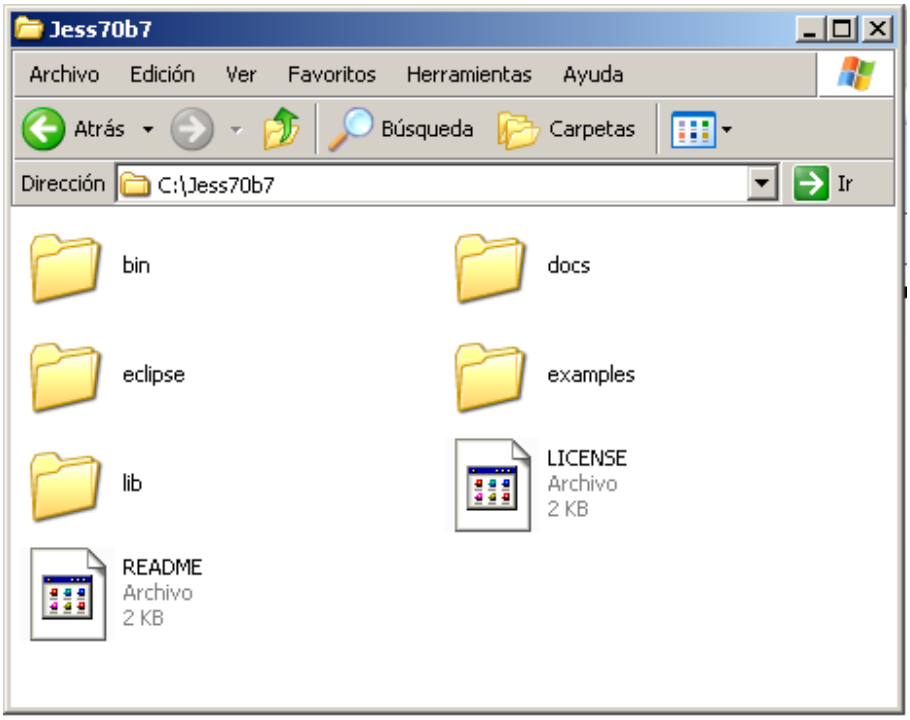
INSTALACIÓN PARA UNA INTERFAZ EN CONSOLA:

Ahora mencionaremos los pasos de instalación para la interfaz en consola de Jess:

- Obtener la carpeta comprimida de la página de JESS.
  Para poder obtener la carpeta se debería registrar en la siguiente pagina.
  http://www.jessrules.com/jess/download.shtml
  Luego escoger la opción para programadores:
  Jess 7.0b7 classes, docs and samples Zid format (Dependiendo de la versión).
  Esta versión es de prueba y tiene un mes de funcionamiento.

- Descomprimir el archivo en un directorio, preferentemente en la raíz de la unidad C:

  Se tendrá la siguiente estructura:



Alguna descripción de los archivos de la carpeta descomprimida de jess:

bin: Contiene el archivo de ejecución para plataformas Windows.

docs: Contiene la documentación de JESS.

examples: Ejemplos.

lib: es el directorio que contiene al motor JESS en un archivo ".jar". Este directorio también contiene el API JSR-94 (javax.rules) en el archivo jsr94.jar.

- Poner una variable de entorno JAVA_HOME a la raíz del jdk:

MiPc → Propiedades → Opciones avanzadas → Variables de entorno → Nueva

En la sección "variables del sistema" se debería escribir:

Nombre de variable: JAVA_HOME

Valor de la variable: C:\Archivos de programa\Java\jdk1.5.0_06

(El directorio puede cambiar**)**

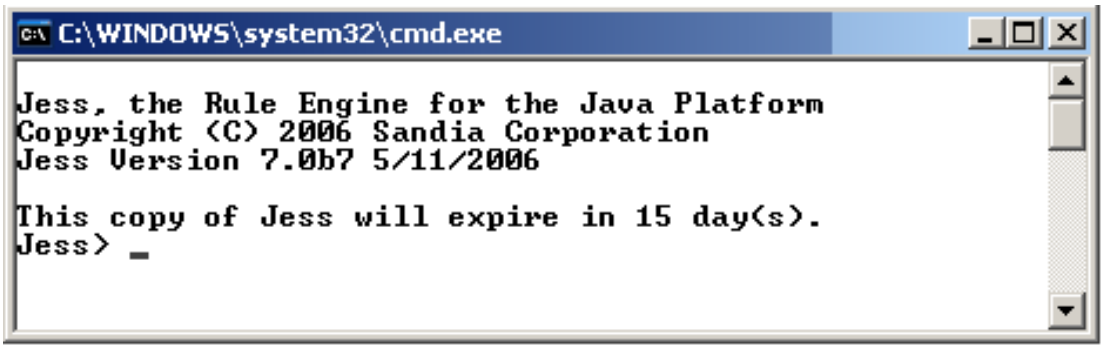
Reiniciar el computador.

- Hacer la prueba de la instalación de Jess.

  Ejecutar el archivo:

  C:\Jess70b7\bin\jess.bat

  Debería tener la siguiente salida:



Para verificar la funcionalidad de nuestra consola, se puede ingresar el siguiente ejemplo:

- Ingresar:
  >(+ 2 2)
  Resultado:
  >4

ELEMENTOS BASICOS

Las palabras que definamos en JESS pueden contener **números**, **letras** (menos la **ñ**) y los símbolos **$*=+/<>_?#.-**. Puedes combinarlos como quieras pero con ciertas restricciones:

- Los literales no pueden comenzar por un número.
- Tampoco pueden comenzar por los símbolos **$**,**?**, o **=** pues tienen cierto significado en JESS como se verá más adelante.

También se debe tener en cuenta que JESS es **sensible a mayúsculas y minúsculas**. Algunas de las **palabras predefinidas** en JESS:

- **nil** (null en JAVA)
- **TRUE**
- **FALSE**

Las **cadenas** de caracteres van entre **“”**.Si quieres que se muestren las comillas dobles debes escribir \ justo antes de cada una de ellas:

“hola” muestra: hola

La barra, por sí sola, no la reconoce JESS. Si intentas mostrarla por pantalla, simplemente la ignora:
“Mi\familia” muestra Mifamilia
Un **comentario** comienza por un punto y coma (**;**) y llega hasta el final de línea:
Esto no es un comentario; Esto si es un comentario

VARIABLES

Las variables son literales que comienzan con **?**. Para asignarle un valor a una variable se usa la función **(bind)**:

(bind ?x “hola”)

Hay otro tipo de variable a la que se le puede asignar varios valores: las **multivariables**. Su nombre comienza por **$?** Y para crear los distintos valores (**multicampos**) se usa la función **(create$)**:

Jess> (bind $?estuche (create$ lapiz boli goma))

Para mostrar el valor de la multivariable:

Jess> $?estuche
(lapiz boli goma)

VARIABLES GLOBALES

Estas variables se borran si sales de JESS o si limpias la memoria con **(reset)**. Si te interesa que tus variables no queden afectadas por la limpieza de memoria usa **variables globales**. Se definen con **defglobal** y su nombre sigue la estructura **?*nombre***. Hay que inicializarlas en el momento en que se crean:

Jess> (defglobal ?*var* = primero)
Jess> TRUE
Jess> ?*var*
primero

Su valor se puede modificar como cualquier otra variable con **bind**. Ahora imagínate que después de haber modificado el valor de tus variables globales vas a hacer un (reset). Qué valor quieres que quede en tus variables, ¿el modificado o el que tenían inicialmente cuando las creaste? Para esto están las funciones **get-reset-globals** y **set-reset-globals**. **set-reset-globals** es una función con un parámetro booleano que le indica al sistema con qué valor quieres que se inicialicen las variables globales después de hacer **(reset)**: **TRUE** si quieres que se inicialicen con su valor inicial y **FALSE** con el último valor asignado. Por ejemplo, siguiendo con la variable ?*var*:

Jess> (bind ?*var* segundo)
Jess> segundo
Jess> (*set-reset-globals *FALSE)
Jess> FALSE
Jess> (reset)
Jess>TRUE
Jess> ?*var*
Segundo
Jess> (*set-reset-globals *TRUE)
Jess> TRUE
Jess> (reset)
Jess>TRUE
Jess> ?*var*
Primero

(get-reset-globals) devuelve el booleano que indica el comportamiento actual de las variables globales, es decir, el último que se indicó en la función **(set-reset-globals)**.

FUNCIONES

En JESS, todo se hace con llamadas a funciones y siempre entre paréntesis. Una función es una lista cuya cabecera es un literal, el nombre de la función. La llamada a una función tiene la siguiente estructura:

(nombreFunción parámetro1 parámetro2 …)

Por ejemplo, dentro de *factorial.clp* está definida la función *fact*, que calcula el factorial del número que se le pasa por parámetro. Puedes cargar el archivo (tienes el enlace al final de la página) y luego llamar a la función:

```
(deffunction fact (?n)
(if (= ?n 0) then 1
else (* ?n (fact (- ?n 1)))
))
```

```
Jess> (batch factorial.clp) Jess> TRUE Jess> (fact 5)
120
```

No hace falta cargar una función cada vez que la quieras utilizar, queda almacenada aunque hagas un **(reset)**, pero si sales del sistema (prompt JESS) entonces sí deberás cargarla de nuevo.

VARIABLES DE LAS FUNCIONES

Si se quiere referenciar a algún parámetro que se le pasa a una función, se usa la variable: **$?argv**. Al resultado de una función, lo que devuelve **return**, se guarda siempre en la variable **?retval**.

FUNCIONES PREDEFINIDAS

JESS tiene un gran número de funciones predefinidas que puedes encontrar en: http://herzberg.ca.sandia.gov/jess/docs/70/functions.html

Además de las nombradas hasta ahora (bind, reset,… son funciones) también hay operaciones matemáticas o los operadores +-*/:

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

Una función muy útil es la de mostrar por pantalla, que ya se vió el el archivo *holaMundo.clp*. La estructura es:

(printout t texto crlf)

Donde:
printout es la función de imprimir.
t dice que lo imprima por la salida estándar, es decir, por pantalla.
crlf hace un salto de línea.

CREAR FUNCIONES

Las funciones se crean con (deffunction) siguiendo la estructura:

(deffunction nombre_función [comentarios] (parámetros)
expresiones
)

(deffunction nombre_función [comentarios] (parámetros)
expresiones
)

Donde *[comentarios]* es una cadena opcional para explicar qué hace la función o para añadir alguna aclaración.

Cuando una función devuelve algo se usa la función:

(return ?resultado)

Imagínate que quieres crear una función pero que según ciertas circunstancias el número de parámetros varía. Esto se puede solucionar nombrando el último parámetro como **multicampo**.

AÑADIR CÓDIGO A UNA FUNCIÓN

Si quieres ejecutar un código antes o después de una llamada a función (ya sea predefinida o que la hayas creado tú) puedes usar la función **(defadvice)**:

(defadvice after/before nombreFunción (expresión))

Cada función sólo puede tener activado **un defadvice**, si ejecutas otro sobrescribirá al primero.

Hay que tener cuidado en las funciones recursivas porque este código se ejecuta con cada llamada a la función.

Para eliminarlo:

(undefadvice nombreFunción)

Ejemplo:

Jess> (deffunction suma (?n1 ?n2) (+ ?n1 ?n2))
TRUE
Jess> (defadvice before suma (printout t "Resultado: " crlf))
TRUE
Jess> (suma 1 2)
Resultado:
3
Jess> (undefadvice suma)
TRUE

Jess> (deffunction suma (?n1 ?n2) (+ ?n1 ?n2))
TRUE

Estructuras

En JESS, las estructuras if, while, for, try y foreach se comportan como funciones, pero funcionan de un modo muy similar a JAVA y a otros lenguajes de programación de alto nivel.

IF

(if (condición) then (acciones1) [else (acciones2)])

Funciona igual que en otros lenguajes, si se cumple *condición* ejecuta *acciones1*, sino ejecuta *acciones2*. *else (acciones2)* es opcional.

Ejemplo:

Jess> (bind ?nota 7.5) 7.5
Jess> (if (< ?nota 5) then (printout t "Estás suspenso." crlf)
else (printout t "Estás aprobado." crlf))
Estás aprobado.

WHILE

(while (condición) [do] (acciones))

Mientras se cumpla condición ejecuta acciones. *[do]* es opcional, de hecho puede dar problemas, así que mejor no lo pongas.

Ejemplo:

Jess> (bind ?contador 1)
1
Jess>(while (< ?contador 4) (printout t ?contador crlf) (++ ?contador))
1
2
3
FALSE

FOR

(for (inicializador) (condición) (incremento) (acciones))

Se inicializa una variable en *inicializador*, mientras cumpla *condición* se ejecutan *acciones* y tras cada pasada del bucle se ejecuta *incremento*. Tanto *inicializador* como *condición* como *incremento* pueden quedar vacíos, pero se deben poner los paréntesis.

Ejemplo:

Jess> (for (bind ?i 1) (< ?i 4) (++ ?i) (printout t ?i crlf))

Jess> (for (bind ?i 1) (< ?i 4) (++ ?i) (printout t ?i crlf))
1
2
3
FALSE

FOREACH

(foreach variable (lista) (acciones))

Lista puede ser una lista creada con **create$** o un iterador de la clase java.util.Iterator, *variable* coge cada uno de sus valores y ejecuta con ellos *acciones*.

Ejemplo:

Jess> (foreach ?alimento (create$ fruta carne pescado verduras) (printout t ?alimento crlf))
fruta
carne
pescado
verduras

TRY

(try (acciones1) [catch (acciones2)] [finally (acciones3)])

Esta función gestiona en *catch* las posibles excepciones que puedan aparecer en *acciones1*. Además se puede poner un cacho de código en *finally* que se ejecutará siempre.

En JESS, a *catch* no se le pueden pasar argumentos, por lo que no se puede pasar una variable de tipo excepción. Puede haber un **catch** o un *finally* o ambos. Aunque no se ejecute nada en *catch* tiene que aparecer la palabra.

Ejemplos:

Jess> (try (open archivo.txt r) catch (printout t "El archivo no se encontró." crlf))
no se encontró

Jess> (try (open archivo.txt r) catch finally (printout t "El archivo no se encontró." crlf))
no se encontró

La variable **?ERROR** devuelve el objeto excepción. Para ver lo que pone:

(printout t (call ?ERROR toString) crlf)

HECHOS NO ORDENADOS

Para generar hechos no ordenados se necesitan unos patrones que actúan como las clases de JAVA, es decir, declaran la estructura de los hechos definiendo los **slots** (campos) que contienen. Estos patrones se definen del siguiente modo:

(deftemplate nombreDeftemplate [extends nombreClase][documento]
[(slot nombreSlot [default|default-dynamic valor] [(type tipo)])]*)

Donde:

- Puede heredar de otro **deftemplate** definido anteriormente (**extends**).
- **default | default-dynamic** indica el valor por defecto del slot. La versión default-dynamic evalúa el valor cada vez que se inserta un nuevo hecho.
- **type** puede ser: ANY, INTEGER, FLOAT, NUMBER, ATOM, STRING, LEXEME u OBJECT.

Ejemplo:

Jess> (deftemplate persona "datos personales" (slot nombre (type STRING)) (slot edad) (slot pais (default España)))
TRUE
Jess> (assert (persona (nombre "Diana") (edad 24)))
<Fact-0>
Jess> (assert (persona (nombre "Cristina") (edad 15) (pais España)))
<Fact-1>

Si quieres que un slot contenga varios valores se define como **multislot**.
Ejemplo:

Jess> (deftemplate persona2 "datos personales" (spot nombre (type STRING)) (slot edad) (multislot telefono) (slot pais (default España)))
TRUE
Jess> (assert (persona2 (nombre "Diana") (edad 24) (telefono 988123456 669123456)))<Fact-0>

Un ejemplo de un hecho que herede de otro:

Jess> (deftemplate trabajador extends persona2 (spot puesto) (slot sueldo))
TRUE

Para guardar un hecho en una variable se hace al crearlo. Por ejemplo:

Jess> (bind ?var (assert (persona2 (nombre "Cristina") (edad 15) (telefono 988123456 669123456))))
<Fact-1>

De este modo se puede modificar alguno de sus parámetros con la función **(modify)**:

Jess> (modify ?var (nombre "Pilar"))

Si quieres introducir varios hechos a la vez, en vez de ir uno por uno con la función **assert**, puedes hacerlo con el constructor **deffacts**. Para que guarde estos hechos debes introducir **(reset)** después de crearlos. Ejemplo:

Jess> (deftemplate mujer (slot nombre)) TRUE
Jess> (deffacts mujeres (mujer (nombre Diana)) (mujer (nombre Cristina)))
TRUEJess> (reset) TRUE
Jess> (facts)
f-0 (MAIN:: initial-fact)
f-1 (MAIN:: mujer (nombre Diana))
f-2 (MAIN:: mujer (nombre Cristina))
For a total of 3 facts in module MAIN.

HECHOS ORDENADOS

Un hecho ordenado es una lista cuyo primer parámetro es el nombre. Que los datos estén ordenados no significa que estén estructurados.

Para crear un hecho: **(assert (lista))**. Esta función asigna un identificador numérico a cada hecho.
Para ver todos los hechos: **(facts)**.
Para borrar todos los hechos: **(clear)**.

Para borrar un solo hecho: **(retract (fact-id identificador))**, donde *identificador* es el número que identifica al hecho. O también **(retract-string "(hecho)")** donde se pasa el hecho que se quiere borrar como una cadena.

Ejemplo:

Jess> (assert (vivienda ocupada))
<Fact-0>Jess> (assert (puerta abierta)) <Fact-1>
Jess> (facts)
f-0 (MAIN:: ((vivienda ocupada))
f-1 (MAIN:: (puerta abierta))
For a total of 2 facts in module MAIN.
Jess> (retract (fact-id 1))
TRUE
Jess> (facts)
f-0 (MAIN:: (vivienda ocupada))For a total of 1 facts in module MAIN. Jess> (retract-string "(vivienda ocupada)")
TRUE
Jess> (facts) For a total of 0 facts in module MAIN.
(reset)

También se borran todos los hechos, pero luego añade un hecho **(inicial-fact)** que JESS usa para sus operaciones. Es mejor no borrar este hecho.

- ***Hechos ordenados :***
  - Ventaja : Se pueden utilizar sin declaración previa
  - Inconveniente : Ningún control sobre el tipo de datos
  - Inconveniente : poco explícito (fuente potencial de errores)
  - Atención : !La posición de un valor puede tener importancia ! **(empleado "Fernandez" "Juan" 27 PATC)**

- **Hechos ordenados :**
  - Instrucción de creación : **(assert <pattern>+)**
  - Sin declaración previa
  - Ejemplo creación: **(assert (objetivo Roby coger cubo)**

!**Las *templates*** se deben declarar antes de utilizarse !

- El orden de los atributos no importa
- Los atributos pueden ser monovaluados o multivaluados

- Un atributo multivaluado se maneja como un hecho ordenado

EL CONSTRUCTOR DEFFACTS

- Se puede definir muchos hechos con el comando deffacts, por ejemplo.

Jess> (deffacts my-facts

"The documentation string"

(foo bar)

(box (location garage) (contents scissors paper rock))

(used-auto (year 1992) (make Saturn) (model SL1)

(mileage 120000) (blue-book-value 3500)

(owners ejfried)))

TRUE

Jess> (reset)

TRUE

Jess> (facts)

f-0 (MAIN::initial-fact)

f-1 (MAIN::foo bar)

f-2 (MAIN::box (location garage)

(contents scissors paper rock))

f-3 (MAIN::used-auto (make Saturn) (model SL1)

(year 1992) (color white)

(mileage 120000)

(blue-book-value 3500) (owners ejfried))

For a total of 4 facts in module MAIN.

CONSTRUCCIÓN DE REGLAS

- Las reglas JESS son como una instrucción if… then pero no es usado de la misma manera.
- Las instrucciones if.. Then son ejecutadas en orden especifico.
- Las reglas JESS son ejecutadas si las partes son satisfechas, haciendo q estas se ejcuten.
- Entonces JESS es menos determinístico.
    - Por ejemplo:

- **Jess> (clear)**
- **Jess> (watch all)**
- **Jess>** (deftemplate person (slot firstName)
  (slot lastName)

  (slot age))

- **Jess>** (defrule welcome-toddlers
  "Give a special greeting to young children"

  (person {age < 3})

  =>

  (printout t "Hello, little one!" crlf))

- **Jess>** (assert (person (age 2)))
- **Jess>** (run)

SIMPLES PATRONES

- Un patrones es un conjunto de parentesis incluyendo el hecho a ser emparejado con la descripción de cero o más campos.

```
Jess> (defrule welcome-toddlers
                    "Give a special greeting to young children"
                    (person {age < 3})
                     =>
                    ((System.out) println "Hello, little one!"))
```

- Se puede declarar una variable para referir a un campo de una plantilla (deftemplate)

```
(person (age ?a) (firstName ?f) (lastName ?l))
```

Podremos utilizar estas variables dentro la misma regla, tanto en la parte derecha como en la izquierda.

- Existen algunos operadores q se usan en un formato infijo como los operadores en JAVA.
- Dentro un simple patrón un símbolo representa el valor de un campo (slot) en la misma plantilla con el mismo nombre
- Por ejemplo emparejar una persona que este entre los 13 a 19 años.

```
Jess> (defrule teenager
          (person {age > 12 && age < 20}
          (firstName ?name))
                =>
          (printout t ?name " is " ?age " years old." crlf))
```

- Cuando se utilizan los campos de la plantilla JESS crea variables de forma automática para q sean usados en la parte derecha de la regla

```
Jess> (defrule same-first-and-last-name
                    (person {firstName == lastName})
                     =>
```

(printout t ?firstName

" " ?lastName "

is a funny name!" crlf))

- Una limitación de los patrones utilizando llaves es q solamente se pueden utilizar en hechos desordenados

PATRONES AVANZADOS

- Se pueden poner restricciones al comparar un
- patrón:
- negación (~) (color ~rojo)
- conjunción (&) (color rojo&amarillo)
- disyunción (|) (color rojo|amarillo)
- También se pueden unir patrones con las
- relaciones lógicas or, and y not
- por defecto, los patrones se unen con and

Ejemplo: (patrones avanzados)

(defrule no-cruzar

(luz ~verde)

=> (printout t “No cruce” crlf)**)**

(defrule precaucion

(luz amarilla | intermitente)

=> (printout t “Cruce con precaución” crlf)**)**

(defrule regla-imposible

(luz verde & roja)

=> (printout t “¡¡MILAGRO!!” crlf)**)**

(defrule regla-tonta

(luz verde & ~ roja)

=> (printout t “Luz verde” crlf)**)**

(defrule precaucion

(luz ?color & amarillo | intermitente)

=> (printout t “Cuidado luz ” ?color crlf)**)**

(defrule no-cruzar

(estado caminando)(or (luz roja)(policia dice no cruzar)(not (luz ?))); sin luz

=> (printout t “No cruzar” crlf)**)**

Ejemplo de Patrones avanzados

- **(cordinate (x ?x&:(> ?x 10)))** ; Creación de la variable ?x el cual debe cumplir la condición ?x > 10.

- **(coordinate (x ?x) (y =(+ ?x 1))) ;** Incremento en una unidad a la vaiable ?x el resultado será asignado a y, por ejemplo:

- **Jess>** (defrule example-3
  (not-b-and-c ?n1&~b ?n2&~c) (different ?d1 ~?d1) (same ?s ?s)

  (more-than-one-hundred ?m&:(> ?m 100))

  (red-or-blue red | blue)

  => (printout t "Found what I wanted!" crlf))

CONSTRUYENDO PATRONES

- Borrar un hecho.
  Jess> (defrule example-5

  ?fact <- (a "retract me")

  =>

  (retract ?fact))

- Se puede llamar en forma directa a los métodos usando el fact-id
  Jess> (defrule example-5-1

  ?fact < - (initial-fact)

  =>

  (printout t (call ?fact getName) crlf)) *TRUE*

  Jess> (reset)*TRUE*

  Jess> (run)

  initial-fact 1

MOTOR DE INFERENCIAS

- El motor de inferencias trata de emparejar la lista de hechos con los patrones de las reglas
- Si todos los patrones de una regla están emparejados se dice que dicha regla está
- activada
- La agenda almacena la lista de activaciones por orden de prioridad
- Para insertar una activación en la agenda, se siguen las estrategias de resolución de conflictos

RESOLUCIÓN DE CONFLICTOS

Salience: es una función que da prioridad a las reglas.

Cada regla tiene una prioridad llamada salience

Cuando una regla es activada, se coloca en la agenda según los siguientes criterios:

1. **Las reglas más recientemente activadas** se colocan encima de las reglas con menor prioridad, y debajo de las de mayor prioridad

2. Entre reglas de la misma prioridad, se emplea la **estrategia configurada** de resolución de conflictos

3. Si varias reglas son activadas por la aserción de los mismos hechos, y no se puede determinar su orden en la agenda según los criterios anteriores, se insertan de forma arbitraria (no aleatoria)

Ejemplo: Salience

Jess> (defrule example-6 (declare (salience -100))

(command exit-when-idle)

=>

(printout t "exiting..." crlf))

Ejemplo: activación de reglas

- hecho-a activa r1 y r2
- hecho-b activa r3 y r4
- añadimos a MT hecho-a y hecho-b en este orden
- Estrategia en profundidad (*Depth*)
- Es la estrategia por defecto
- Agenda r3, r4, r1, r2
- Estrategia en anchura (*Breadth*)
- Agenda r1, r2, r3, r4

ENCADENAMIENTO ADELANTE Y ATRÁS

Las reglas anteriormente vistas tienen que ser tratadas como un instrucción if.. then.

Si se cumple la parte izquierda entonces se ejecuta la parte derecha.

Prolog y sus derivados soportan encadenamiento hacia atrás. Pero muchas veces no necesariamente podemos tener las condiciones previas.

Primero se debe crear una plantilla con encadenamiento hacia atrás.

```
Jess> (deftemplate factorial
                (declare (ordered TRUE)
                                        (backchain-reactive TRUE)))
```

Alternativamente se debe poder usar la función.

```
Jess> (do-backward-chaining factorial)
```

Construir una regla para emparejar la plantilla.

```
Jess> (defrule print-factorial-10
      (factorial 10 ?r1)
       =>
       (printout t "The factorial of 10 is " ?r1 crlf))
```

El compilador inserta un hecho

```
(need-factorial 10 nil)
```

Se debería construir una regla para ese hecho.

```
Jess> (defrule do-factorial
          (need-factorial ?x ?)
          =>  (bind ?r 1)
              (bind ?n ?x)
              (while (> ?n 1)
                  (bind ?r (* ?r ?n))
                  (bind ?n (- ?n 1)))
              (assert (factorial ?x ?r)) )
```

Jess>(reset)

Jess>(run)

REFLEXIÓN

- Se pueden manipular todos los objetos de java directamente desde JESS.
- Se puede hacer virtualmente todo desde el código JESS, excepto definir nuevas clases.
- Por ejemplo:

Jess> (bind ?ht (new java.util.HashMap))

<Java-Object:java.util.HashMap>

Jess> (call ?ht put "key1" "element1")

Jess> (call ?ht put "key2" "element2")

Jess> (call ?ht get "key1")

"element1"

- La conversión de tipos es libre por JESS.
- Los objetos JAVA que no pueden estar representados como un tipo JESS son llamados Java objects .

PLANTILLAS

- Son como clases pero sin herencia
- Permiten representar hechos no ordenados
- Se definen como hechos (*deffacts/assert*)
- Emparejamiento:
  (<plantilla> (<atributo> <patrón1>)

  (<atrib2> <patrón2>)...)

Ejemplo:

(deftemplate persona

(slot nombre (type SYMBOL))

(slot edad (type NUMBER)

(range 0 99)(default 20))

(slot estado (type SYMBOL)

(allowed-symbols soltero casado viudo)

(default soltero))**)**