FAQ de CLIPS

FIB - UPC

Curs 2023/2024 2Q

UPC
FIB Facultat d'Informàtica de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

En la elaboración de este documento han intervenido:

Lluís Alemany Puig
Javier Biosca Ruíz de Ojeda
José Camallonga González
Jordi Chacón Chacón
Adrià Figuera Puig
Martí Fornés Estarellas
Daniel Golobart Castellote
Marina Grigoreva
Borja Jara García
Brian Jiménez Gracía
Ignacio Llatser Martí
Victor Lloveras Díaz
Isaac López Amat
Víctor Martínez Jurado
Lluís Monsalve Carrasquilla
Enric Munné Hernández
Jorge Muñoz Gama
Ivan Navarro González
Manuel Parrilla Gutierrez
Marcos Pereira Varela
Pere Sivecas Gibert
Lluís Suñol Juliachs

El responsable de la edición/corrección/ampliación del documento es Javier Béjar Alonso (`bejar@cs.upc.edu`)

Departament de Ciéncies de la Computació

Facultat d'Informática de Barcelona

Universitat Politècnica de Catalunya

Imágenes generadas con Stable Diffusion

Primera edición, Septiembre 2008

Esta edición, Febrero 2024

Índice general

1. Introducción

Este documento es el resultado de la recopilación de los documentos escritos durante el desarrollo de la práctica de sistemas basados en el conocimiento el cuatrimestre de otoño del curso 2007-2008.

El objetivo de estos documentos era recoger las dudas frecuentes que se encuentra un alumno durante el desarrollo de la práctica de SBCs y consejos prácticos que puedan ayudar en el desarrollo.

Este documento recoge principalmente dudas frecuentes sobre el entorno CLIPS y su lenguaje de programación.

2. Protègè

Para saber como funciona Protègè usad la ayuda de la aplicación y el material correspondiente, estas cuestiones se refieren a como integrar lo que se desarrolla en Protègè con CLIPS

2.1 Como exporto una ontologia de Protégé a CLIPS?

Para exportar la ontología simplemente hay que usar la opción del menú File → Save As y escoger el formato `OWL/XML` o `Turtle`.

Una vez grabado el fichero tendréis que usar la librería python `owl2else` que deberéis instalar primero en la distribución de python que estéis utilizando. Lo podéis hacer usando el comando `pip` de esta manera:

```
pip install owl2else --user
```

Una vez instalado podéis usar el comando `owl2clips` que recibe como parámetros el fichero de entrada, el fichero donde se dejará la conversión y el formato que tiene el fichero de entrada, por ejemplo, si está en formato turtle

```
owl2else --input ontologia.ttl --output ontologia.clp --format turtle
```

o si esta en formato owl

```
owl2else --input ontologia.owl --output ontologia.clp --format xml
```

Si habéis usado la versión web de Protègè y no lo habéis configurado para que use los nombres que les dais a los conceptos como los símbolos en la ontología tendéis que añadir el flag `--label`.

2.2 ¿Cómo genero un gráfico de la ontología?

Protègè tiene dos plug-ins que pueden usarse para generar un gráfico de la ontología. Adicionalmente la librería `owl2else` tiene el comando `owl2plot` que permite generar gráficos de

la ontología completa o de una clase padre de la jerarquía. Para poder usarlo hay que tener instalado Graphviz (`https://graphviz.org/`).

Tiene los parámetros:

- `--input`: Fichero de la ontología
- `--output`: Fichero de salida
- `--format`: Formato en el que esta la ontología (xml, turtle)
- `--gformat`: Formato del fichero de salida (pdf, png, jpg, svg)
- `--labels`: Usar los nombres que aparecen en las etiquetas de las clases en lugar de los símbolos que le asigna Protègè
- `--dot` (opcional): No borra el fichero `.dot` que se genera para poder usarlo con graphviz
- `--topclass` (opcional): Solo representa una clase padre de la jerarquía (incluye las relaciones con otras clases que tenga)

2.3 Per què em dóna error CLIPS al tenir una instancia que hereta de més d'una classe?

Protégé permet l'herència múltiple en instàncies, però CLIPS no. En CLIPS només poden tenir herència múltiple las classes, però no les instàncies. Això es així perque el lenguatge de CLIPS es orientat a objectes, no es realment un llenguatge d'ontologíes.

La manera més senzilla de simular-ho, si les clases no tenen ancestres comuns, és tenir classes que representin aquesta herència múltiple i fer que les instàncies pengin d'elles, que es com es faria a un llenguatge orientat a objectes.

En el cas de tenir ancestres comuns no es pot fer, l'unica manera es duplicar la instancia amb noms diferents i penjar-les de les clases on es vol tenir-la.

De totes maneres, el CLIPS permet recuperar fàcilment les instàncies que tenen un valor específic en un slot, pel qual el més senzill és tenir una única jerarquia i incloure altres possibles classificacions com a slots, encara que obviament perdrem la posibilitat de fer servir una jerarquia de valors per aquest.

3. CLIPS

3.1 Instalación del CLIPS

3.1.1 ¿Cómo instalar CLIPS?

La herramienta CLIPS es multiplataforma y esta disponible para los tres S.O más utilizados: Microsoft Windows, Linux y Mac OS X. Usaremos la versión 6.40.

3.1.2 ¿Cómo instalar CLIPS en Windows?

Primero de todo debemos bajarnos el programa de la página oficial de CLIPS, accesible desde este link https://clipsrules.net/. Desde el área de descarga os podreis bajar el instalador para windows.

3.1.3 ¿Cómo instalar CLIPS en Linux?

En muchas distribuciones de Linux CLIPS está disponible como un paquete adicional, pero no corresponde con la última versión. Desde la página https://clipsrules.net/ se pueden bajar los fuentes y compilarlos. Esta versión no tiene interfaz gráfica.

También está la posibilidad de utilizar el emulador de windows WINE. Una vez instalado el emulador se puede instalar CLIPS y este funciona igual que sobre windows.

3.1.4 ¿Cómo instalar CLIPS en Mac OS X?

Primero de todo debemos bajarnos el programa de la página oficial de CLIPS, accesible desde este enlace http://www.clipsrules.net . Desde el área de descarga podeis bajar el instalador para MAC OS X.

3.1.5 Entorn finestres VS Consola de comandes

Encara que l'entorn de finestres és molt més intuïtiu, CLIPS a la seva versió en línea de comandes és molt més ràpid (segurament perquè no ha de mostrar en tot moment l'agenda de fets, les instancies, les regles, etc..).

La diferencia de velocitat és molt notable, en cas de problemes de rendiment o de sensació que l'entorn de finestres es queda penjat (o directament es tanca sense previ avís) recomanem provar la versió en línea de comandes.

3.2 Los hechos

3.2.1 ¿Qué es un hecho en CLIPS?

Conceptualmente, un hecho representa un dato abstracto del cual almacenamos valores. Se puede ver como una lista de campos, donde el primer campo hace referencia al "dato" al cual le asignamos una lista de valores que le suceden, teniendo una relación entre sí.

Formalmente, el primer campo es un símbolo y los campos restantes son valores.

```
(símbolo  valor1 valor2  ...  valorN)
```

3.2.2 ¿Qué es un hecho ordenado (order Facts)?

Un hecho ordenado tienen formato libre, por lo tanto no existe restricción alguna en el orden de los campos. Los campos de un hecho ordenado pueden ser de cualquier tipo primitivo de datos, excepto el primero, que debe ser un símbolo. Como ejemplo tenemos:

```
1 (hermanos Antonio Javier Carlos)
2 (padre Pedro David)
```

3.2.3 ¿Qué es un hecho no ordenado (deftemplates facts)?

A través de los hechos no ordenados podemos abstraernos de la estructura de un hecho, asignado un nombre a cada campo (slots).

```
1   (deftemplate   nombre-template   "comentario"
2        (slot nombre-slot  (tipo-del-slot))
3        (multislot nombre-slot  (tipo-del-slot))
4   )
```

Por ejemplo:

```
1 (deftemplate avión
2         (slot nombre_avion (type STRING))
3         (slot compania (type STRING))
4         (slot numero_plazas (type INTEGER) (default 100))
```

3.2.4 ¿Puedo tener un vector como slot?

Si, debe declararse como multislot.

3.2.5 ¿Puedo asignar propiedades a los slots?

Sí, las más importantes son:

Type: El tipo de dato primitivo que contiene (en mayúsculas).

Allowed-X: Permite especificar un conjunto de valores permitidos. Siendo `X` un tipo de datos primitivo (en minúsculas y plural) → si el slot toma un valor del tipo `X` debe pertenecer a los valores permitidos. `X` puede ser `values` especificando entonces todos los valores posibles que puede tomar el slot independientemente del tipo de dato.

Range: Especifica un rango para slots numéricos.

Cardinality: Números mínimo y máximo de elementos que puede tener un multislot.

Default: Valor por defecto. Si ponemos `?NONE` como valor por defecto significará que no se podrá crear la instancia a no ser que se especifique el valor de ese slot.

```
(deftemplate estudiante
 (slot edad (type INTEGER) (range 0 99))
 (multislot asignaturas (type STRING) (cardinality 1 5))
 (slot num-carreres (default 1))
 (multislot notas (allowed-strings "MH" "NP"))
)
```

3.2.6 ¿Qué debo utilizar, hechos ordenados o no ordenados?

Como se puede intuir si tenemos un conjunto de hechos los cuales conceptualmente pertenecen al mismo ámbito, será una solución más elegante agruparlos a través de un deftemplate. En caso que tengamos que algún hecho aislado, será mas sencillo declararlo como un hecho ordenado.

3.2.7 Fets o Objectes?

Veient la definició de deftemplate podem confondre'ns a l'hora de decidir si fer servir fets o fer servir objectes.

Deixant de banda altres diferències ja comentades (les classes suporten jerarquia de classes, herència, etc..), els objectes es fan servir per representar el coneixement (mitjançant un conjunt d'instàncies).

Així que, generalment, farem servir classes i objectes per representar coneixement, mentre que farem servir els fets per portar control sobre l'execució del problema.

3.2.8 Creación de hechos (assert/deffacts)

Si queremos crear un solo hecho utilizaremos el comando `assert`. Por ejemplo: `(assert (casa roja))`

Si por el contrario queremos crear una estructura ordenada de hechos utilizaremos deffacts:

```
(deffacts mishechos (casa roja) (pelota azul))
```

3.2.9 Modificar un fet

Per modificar un fet (per exemple, molt útil si estem treballant amb deftemplates) ho podem fer mitjançant `(modify <INSTANCIA> <FETS>)`.

Per exemple, si tinguéssim el deftemplate persona

```
(deftemplate persona
 (slot nom)
 (slot edad)
```

```
4    (slot dni)
5   )
```

I haguéssim afegit el fet:

I el fet:

```
(assert (aniversari "46974431"))
```

Podríem tenir la següent regla:

```
1   (defrule aniv
2     ?aniversari <- (aniversari ?dni)
3     ?persona <- (persona (edat ?edat)(dni ?dni))
4     =>
5     (retract ?aniversari)
6     (modify ?persona (edat (+ 1 ?edat)))
7   )
```

La qual s'activarà si hi ha un fet aniversari d'una persona. Fent servir el dni indicat al fet aniversari, busquem el fet persona que té el mateix dni. Deixem el valor edat a la variable `?edat` per poder treballar amb ella més endavant.

Hem agafat també la direcció del fet aniversari per tal de poder esborrar-lo de la base de fets. Això s'ha fet per dues raons:

Primer, per mantenir més neta la base de fets. Segon, i més important: Com que hem canviat el fet, la regla tornarà a fer unificació amb el fet (perquè ara és diferent) amb la regla aniversari, amb el que entraria dintre d'un bucle sense fi on s'aniria incrementant l'edat de la persona.

Un cop hem esborrat el fet aniversari, incrementem en 1 el valor del slot `edat` de la persona, fent servir la variable `?edat` que hem declarat a la part esquerre de la regla.

El comportament de `modify` es fer un retract del fet a la base de fets i després fer un assert amb els nous valors. Si volem canviar mes d'un slot del fet es convenient canviar-los tots a la vegada. La raó es que cada vegada que fem un `modify` l'apuntador al fet es modifica i l'apuntador que tenim a la variable que hem fet servir ja no es vàlid.

Si no temim mes remei que fer la modificació del fet pas a pas, s'ha de saber que la crida a `modify` retorna l'apuntador al nou fet. Si el guardem a altra variable (o a la mateixa) podrem continuar modificant el fet amb la nova referència.

3.2.10 Com elimino un fet de la base de fets?

Per eliminar un fet de la base de fets ho podem fer amb la paraula clau `(retract <FET>)`, on `<FET>` és un punter al fet.

Per obtenir aquest fet, ho podem fer fent servir `<VAR> <- <FET>` a la part esquerre de la regla.

Per exemple:

```
1   (defrule accionsvent
2    ?vent <- (vent ?tipus)
```

```
3    =>
4    (switch ?tipus
5       (case poc then (assert (accio persiana0)))
6       (case normal then (assert (accio persiana50)))
7       (case molt then (assert (accio persiana100)))
8    )
9    (retract ?vent)
10  )
```

Afegirà fets per realitzar les accions pertinents i esborrarà el fet de la base de fets.

3.3 Variables

3.3.1 Variables a CLIPS

No cal declarar una variable (ni el seu tipus), només cal afegir un interrogant al començament d'un simbol per indicar que és una variable.

CLIPS s'encarrega del control de tipus depenent del que assignem a la variable.

Per exemple, `?testvar` correspondria a la variable `testvar`. Si per exemple, assignéssim l'enter `3` a `?testvar` CLIPS tractaria la variable com una variable entera.

3.3.2 Com assignar un valor a una variable?

Per assignar un valor a una variable es fa servir el mètode `(bind)`, amb la forma:

```
(bind <VARIABLE> <VALOR>)
```

On `<VALOR>` pot ser un valor o un mètode que retorni un valor. Per exemple, si volem assignar un 4 a la variable `?var`:

```
(bind ?var 4)
```

O per exemple, podríem assignar a la variable `?var` la suma de 2 més 2:

```
(bind ?var (+ 2 2))
```

3.3.3 ¿Cómo declarar variables globales?

Para crear una variable global hay que meterla dentro de la construcción defglobal. La declaración de la variable seguirá el modelo: `?*nombreVariable* = expresión`

Por ejemplo:

```
(defglobal ?*presupuesto* = 0)
```

Pueden aparecer en la parte izquierda de las reglas si no son utilizadas para asignar un valor y su cambio no activa reglas, pero no pueden ser parámetros de funciones ni métodos.

3.3.4 ¿Cómo trabajar con variables, sumando, restando, ... por ejemplo contadores?

La mejor forma es con variables globales, se definen así `(defglobal ?*x* = 0)`, puede ser el valor que queramos, tanto número, string, ... Y si queremos sumarle 10 por ejemplo:

```
(bind ?*x* (+ ?*x* 10))
```

Las variables locales, como por ejemplo `?x`, sólo tienen valor mientras dura su ejecución y eso suele ser en esa misma línea o en un bucle. Si le damos valor a `?x` y luego preguntamos por ella, nos dirá que no existe.

3.3.5 Quina diferència hi ha entre les variables del tipus `?nom_variable` i les del tipus `$?nom_variable`?

Les primeres contenen un sol valor, i les segones en poder contenir més d'un (o cap).

3.3.6 Què significa l'interrogant (?) quan va sol?

És una variable anònima. Pots utilitzar `?` sense posar-hi cap nom si no t'interessa el valor en concret. Per exemple:

```
(Primer filtre_preu ?)
```

Aquesta condició significa "si el fet `(Primer filtre_preu)` té algun valor". O sigui, que si tinguéssim el fet `(Primer filtre_preu ok)` es compliria la condició (i si en comptes de "ok" fos "patata" també es compliria). En canvi, si el fet fos únicament `(Primer filtre_preu)` no es compliria la condició.

3.3.7 Què significa el dòlar ($) quan va sol?

És com l'interrogant, però admet més d'un valor. Per exemple:

```
(Primer filtre_preu $ ok)
```

donarà cert pels fets de l'estil `(Primer filtre_preu patata ok)`, `(Primer filtre_preu hola ok)`, `(Primer filtre_preu patata hola ok)`, `(Primer filtre_preu ok)`. És a dir, no importa els valors que hi hagi entre `filtre_preu` i ok, si al final hi ha un ok.

3.3.8 ¿Como puedo ver los hechos presentes en un momento determinado?

Abriendo la Facts Window en Window de la barra de menú.

3.4 Las reglas

3.4.1 ¿Cómo construir una regla?

Para construir una regla utilizaremos la construcción defrule:

Sintaxis:

```
(defrule <nombre-regla> [<comentario>]
  [<declaración>]
  <elemento-condición>* ; Parte izquierda (LHS)
   =>
  <acción>*) ; Parte dcha. (RHS) de la regla
```

Una regla consta de un conjunto de condiciones (antecedente), también denominados elementos condicionales (EC) o parte izquierda (LHS), y de un conjunto de acciones (consecuente), también denominado parte derecha de la regla (RHS). La regla se activará siempre que se satisfagan todos los EC mediante hechos o instancias que los cumplan.

Si se introduce en la base de reglas una nueva regla con el mismo nombre que el de una existente, la nueva regla reemplazará a la antigua.

Si una regla no tiene parte izquierda, es decir, no tiene elementos condicionales, entonces el hecho `(initial-fact)` actuará como el elemento condicional para ese tipo de reglas, y la regla se activará cada vez que se ejecute un comando reset.

3.4.2 ¿Cómo gestionar la parte izquierda (LHS) de una regla?

Declarar hechos

Si no se especifica ninguna condición la regla se activará siempre. Una condición puede ser un simple hecho (Ej. `vegetariano`) o un hecho ordenado con unos valores concretos (Ej. `(persona (nombre Juan))`). Si lo que queremos es obtener el valor de un hecho insertaremos una variable en su lugar (Ej. `(persona (nombre ?n))`). Entonces entrará en la regla para cada valor posible que pueda asignar a la variable `?n` a partir de los hechos.

Si queremos indicar cualquier valor utilizaremos el símbolo `?` y para cualquier lista el símbolo `$?` (Ej. `(evento ?) (estilo $?)`). También existe la opción de guardar en una variable una referencia al objeto. Ello se consigue poniendo `var <-` a su izquierda (Ej. `?p <- (persona (nombre Juan))`). Se puede usar por ejemplo para pasárselo de parámetro a una función llamada en la RHS.

Establecer condiciones

Además podemos especificar restricciones adicionales sobre los valores que puede tomar un campo de un hecho. Existen tres tipos principales de restricciones: Conectadas (por orden de precedencia o prioridad)

not (`~`): Evita que cierto campo cumpla determinada restricción.

and (`&`): Combina dos restricciones conjuntivamente.

or (`|`): Combina dos restricciones disjuntivamente.

De predicado (`:`) : Obliga a que cierto campo cumpla determinada condición (especificada por el predicado).

Por valor devuelto por una función `=`: Permite llamar a una función y utilizar el valor devuelto para restringir el valor que pueda tomar un campo de un hecho.

Se puede comprobar si se cumple una determinada condición mediante el constructor `test`. Dentro de `test` se pueden hacer llamadas a cualquier función o predicado, ya sea primitiva de CLIPS o definida por nosotros.

3.4.3 Combinar elementos

Los diferentes elementos de la parte izquierda van implícitamente unidos mediante una `AND`, que indica conjunción. No obstante podemos hacerlo explícitamente mediante `(and (cond1) (cond2))`, aunque suelen utilizarse para anidar `ANDs` dentro de `ORs`.

`OR` Sirve para hacer una disyunción `(or (cond1) (cond2))`. Es equivalente a escribir varias reglas en las que cada una de ellas tenga uno de los componentes del OR en su parte izquierda, y la misma parte derecha.

`NOT` Sirve para detectar si no existe un determinado hecho (Ej. `(not (estilo sibarita))`).

`exists` Cuando a un conjunto de elementos se le antecede exists, se comprueba que hay al menos un hecho en la base de hechos que cumpla la condición (Ej. `(exists(bebida ?precio: (< ?precio 5)))`)

`forall` Permite comprobar si todos los hechos que satisfacen un determinado patrón, cumplen una serie de condiciones (Ej. `(forall(plato (estilo moderno))`)

3.4.4 ¿Cómo obtener la dirección de hechos o instancias situados en LHS?

En algunas ocasiones, vamos a necesitar realizar algún tipo de acción en la parte derecha de las reglas sobre hechos o instancias que cumplan unas determinadas condiciones. Pero para ello necesitaremos primero tener acceso a estos elementos para después poder operar sobre ellos. Esto lo conseguiremos guardando la dirección del elemento en una variable haciéndolo del siguiente modo:

```
?variable_direccion <- (elemento)
```

Pongamos un ejemplo para ver realmente el funcionamiento y uso de esta posibilidad que CLIPS nos ofrece:

```
1 (defrule imprime-area
2   ?instancia <- (object (is-a Rectangulo))
3   =>
4   (printout t "Area del rectangulo " (instance-name ?instancia)
5                                ":" (send ?instancia calcula-area) crlf)
6 )
```

Cuando se ejecuta esta regla, se imprimirán las áreas de todas las instancias de la clase `Rectangulo`. En la parte izquierda de la regla, lo que hacemos es guardar en la variable instancia la dirección de la instancia de `Rectangulo` que estamos tratando en ese momento. La regla se ejecutará para todas las instancias almacenadas en la memoria de trabajo que cumplan la condición (`is-a Rectangulo`), es decir, para todas las instancias de la clase `Rectangulo`. En la parte derecha de la regla, donde se ejecutan las acciones, podemos ver que se imprimirá, para cada instancia, lo siguiente: `Area del rectangulo [nombre_instancia]: área del rectángulo`. La función `instance-name` nos devuelve el nombre de la instancia que le pasamos por parámetro. Además, hemos llamado al gestor de mensajes `calcula-area` para que nos devuelva el area del rectángulo.

3.4.5 ¿Cómo hacer sentencias or en la izquierda de las reglas?

Con `(or ...)`. Si por ejemplo tenemos dos hechos `(a)` y `(b)` y queremos ejecutar la derecha de la regla cuando se cumpla uno de los dos, tendremos que hacer `(or (a) (b))`. Atención porque la parte derecha se ejecutará tantas veces como veces se cumpla cada uno de los hechos, si para nosotros existe tanto `(a)` como `(b)`, la parte derecha se ejecutará dos veces. Para que sólo se usa una vez, podemos hacer combinaciones de or y and (funciona igual) o usar un patrón, por ejemplo `(persona Juan|Pedro)` sólo se ejecutaría una vez.

3.4.6 Com afegir una condició a la part esquerre d'una regla?

A vegades és necessari que una regla s'instanciï si es produeix una condició especial (i no un fet). Per fer això, farem servir la comanda `(test <CONDICIO>)`.

Per exemple, imaginem que la nostre base de fets compte amb un fet `(preu ?valor)` amb un preu determinat. Imaginem que volem una regla que s'instanciï quan aquest valor és més gran que 30.

Per fer-ho, fent servir tot el que hem explicat,

```
1 (defrule preucar
2  (preu ?preu)
3  (test (> ?preu 30))
4  =>
```

```
5   (printout t "El preu es mes gran que 30")
6  )
```

És a dir, si el preu és més gran que 30 s'imprimirà un missatge per pantalla.

Aquestes condicions poden contenir ANDs i ORs de varies condicions.

Per exemple,

```
1  (defrule preulimit
2    (preu ?preu)
3    ?factminim <- (preuminim ?minim)
4    ?factmax <- (preumaxim ?maxim)
5    (test (and (> ?preu ?minim) (< ?preu ?maxim) ))
6     =>
7    (retract ?factminim)
8    (retract ?factmax)
9    (printout t "El preu està dintre del rang")
```

On comprovarem si el preu està dintre d'un rang determinat i en aquest cas, esborra els fets que contenen el màxim preu i el mínim preu i mostra un missatge per pantalla.

3.4.7 Puede una misma regla ejecutarse más de una vez?

Es muy importante entender que cada regla se ejecutará una vez por cada combinación de valores en su LHS.

```
1  (defrule regla
2     (color ?col)
3     (peso ?pes)
4   =>
5  ...
```

Si tengo tres colores y dos pesos asertados la regla se ejecutará 6 veces, una con cada combinación de color y peso.

3.4.8 Com puc definir l'ordre de les regles?

La idea d'un sistema basant en regles és no definir un ordre lineal d'aquestes (costa deixar de pensar de forma iterativa)

En el cas de necessitar que una regla s'executi després d'una altre, pots fer servir Fets: la regla anterior crea un fet, que la segona regla tindrà a la part esquerra:

```
1  (defrule pregunta1
2   =>
3   (assert (pregunta1Feta))
4   ....
5  )
6
7  (defrule pregunta2
8   (pregunta1Feta)
9   =>
10  ....
11 )
```

I finalment, pots usar la propietat `salience`. Aquesta propietat indica la prioritat d'execució de les regles (les regles amb salience majors s'executaran abans). Pot prendre valors positius i negatius entre -10.000 y +10.000. Per defecte les regles tenen salience 0:

```
(defrule pregunta1
 (declare (salience 2))
 =>
 ....
)

(defrule pregunta2
 (declare (salience 1))
 =>
 ....
)
```

El valor indicado en la propiedad `salience` puede ser el número directamente o una expresión, por ejemplo `variable*10`, podemos usar `variable` para elegir que regla se ejecutará.

Las prioridades deben ser usadas solo para determinar el orden en el que se disparan algunas reglas sobre otras, no para fijar el flujo de control ya que la principal ventaja de un sistema basado en reglas es la representación declarativa del conocimiento y de esta forma, abusando de las prioridades, conduce a un sistema procedimental.

3.4.9 Com forçar que una regla sigui la primera?

Per que aquesta regla sigui la primera ha de tenir el salience més gran de totes.

```
(defrule presentacio
 (declare (salience 20))
 =>
 (printout t "------ Benvingut -----" crlf)
)
```

3.4.10 Vull guardar certs valors que es fan servir durant l'execució de diferents regles. Com ho puc fer?

Tens principalment dues alternatives. Si el valor és un tipus bàsic, pots fer servir una variable global, amb la sintaxi següent:

```
(defglobal ?*nom_variable* = valor)
```

Aquesta sentència no va a dins de cap funció ni regla, es posa sola a la part que creguis convenient del codi. Vegem un exemple:

```
(defglobal ?*preu_minim_primer* = 1000)
```

Per fer-la servir es fa exactament de la mateixa manera que amb una variable normal, però has de posar els dos asteriscs. Per exemple:

```
(bind ?*preu_minim_primer* ?curr-preu)
```

Per altra banda, si el valor o valors que vols guardar han de ser instàncies, una bona estratègia pot ser utilitzar un deftemplate. És útil sobretot per anar guardant solucions temporals, o llistes d'elements que poden formar part de la solució.

Per utilitzar-ho has de seguir els passos següents.

1. Definir el deftemplate:

```
(deftemplate nom_del_deftemplate
  (multislot nom_multislot)
   ...
  (slot nom_slot)
)
```

Amb un exemple es veurà més clar:

```
(deftemplate llista-plats
 (multislot primers)
 (multislot segons)
 (multislot postres)
)
```

2. Inicialitzar el deftemplate

```
(defrule
  [condicions vàries]
  ; La condició base és que llista-plats no s'ha inicialitzat encara.
  (not (llista-plats))
=>
  ; ?llista conté els valors de llista-plats (que de moment estan buits)
  (bind ?llista (assert (llista-plats)))
```

3. Assignar-li valors

```
(bind ?segons_plats (find-all-instances ((?plat Plat)) TRUE)
(modify ?llista (segons ?segons_plats))
```

4. Recuperar els valors

```
(defrule
  [condicions vàries]
  ?llista <- (llista-plats (segons $?segons_plats))
=>
  ...
```

Ara, la variable `$?segons_plats` conté tots els elements de segons.

3.5 Las clases/instancias

3.5.1 Tengo que representar la jerarquía de un frame ¿Es preferible añadir las subclases a la ontología o bien diferenciarlas mediante un slot en la superclase?

Las dos formas son válidas, pero es recomendable representarlas en clases separadas cuando se vayan a buscar instancias de ese tipo, ya que así no tendremos que hacer un recorrido por todas las de la superclase. No obstante, si sólo se trata de una característica puntual del frame y no vamos a hacer búsquedas de una exclusiva subclase entonces será más eficiente añadir un slot debido a su mejor accesibilidad.

3.5.2 ¿Qué es un objeto?

Podríamos decir que un objeto, es un hecho no ordenado con herencia.

3.5.3 Como se definen las clases?

Para definir las clases usaremos el constructor defclass:

```
(defclass  <nombre>  [<comentario>]
 (is-a  <nombre-de-superclase>)
 [(role concrete | abstract)]
 [(pattern-match reactive | non-reactive)]
 <slot>* ;;; definición de los atributos de la clase
<documentación-handler>*
```

En caso de desear que la nueva clase herede las propiedades de otra ya existente, el nombre de esta deberá especificarse despues de `is-a`. El rol determinará si la clase es concreta (se puede instanciar) o abstracta (destinada a ser superclase). El `pattern-match` debe declararse como reactivo si se desea que las instancias de la clase puedan unificar con los elementos objeto de las reglas (símbolo `<-`). Los `slots` corresponden a los atributos de la clase y los `handlers` a las funciones de ésta. Los handlers se pueden declarar en la declaración de la propia clase, pero se deben definir externamente. Dado que no aporta ninguna ventaja declararlos es recomendable definirlos directamente. Nótese que todo aquello que está entre corchetes (`[]`) constituye información adicional que no es obligatorio especificar.

```
1  (defclass Persona
2   (is-a Ser_Vivo)
3   (role concrete)
4   (single-slot edad
5    (type INTEGER)
6    (range 1 99)
7    (cardinality 0 1)
8    (create-accessor read-write)
9   )
10  (multislot amigos
11   (type INSTANCE)
12   (allowed-classes Persona)
13  )
14 )
```

El comportamiento y declaración de los slots es idéntico al de los slots de los hechos ordenados. Los atributos de tipo `INSTANCE` (referencian otras instancias) estan en formato `INSTANCE-NAME` .

3.5.4 Com consulto una instància a partir d'una regla?

Podem fer servir la part esquerre d'una regla per trobar instàncies. L'ús és molt semblant a trobar el punter a un fet, fent servir

```
(object [(is-a <NOM_CLASSE>)|(name <NOM_INSTANCIA)|(<NOM-SLOT> <VALOR/VARIABLE))]*)
```

Per exemple, si volem que una regla s'instanciï si tenim instanciat un plat de la classe Plat, podriem fer servir:

```
(defrule tenimplat
 ?plat <- (object (is-a Plat))
=>
(print t (send ?plat get-nom) crlf)
```

La regla imprimirà el nom del plat que trobi. És important notar que s'activarà per cada instància de plat que tinguem a la base de coneixement.

Podem consultar els valors de la instancia fent servir condicions a la part esquerre de la regla. Per exemple, si només volguéssim els plats amb un preu més gran que 30:

```
(defrule platCar
 ?plat <- (object (is-a Plat) (Preu ?p))
 (test (> ?p 30))
 =>
 (print t (send ?plat get-nom) " es car." crlf)
```

S'activarà per tots els plats amb un preu major a 30.

S'ha d'anar amb cura per aquest tipus de regla, ja que si la base de coneixement és molt amplia pot acabar derivant amb problemes de memòria.

Per exemple, si en comptes de preguntar per una instància de Plat, preguntéssim per dues instancies de Plat:

```
(defrule platsCars
 ?plat1 <- (object (is-a Plat) (Preu ?p1))
 ?plat2 <- (object (is-a Plat) (Preu ?p2))
 (test (> ?p1 30))
 (test (> ?p2 30))
 =>
 (print t (send ?plat1 get-nom) " i " (send ?plat1 get-nom)   " són cars."
crlf)
)
```

Faria unificació amb totes les combinacions possibles (amb repeticions) de dos plats de tota la base de coneixement. Això es podria anar ampliant fins fer-se intractable.

Si tenim d'alguna manera el nom de la instancia podem fer:

```
(defrule platPicant
 (cuina (plat ?p))
 (object (name ?p) (picant ?pi))
 =>
 (print t " picant " ?pi crlf)
)
```

Això es pot servir per crear condicions que facin servir noms d'instancies que tenim almacenats a slots d'altres instancies, per exemple:

```
(defrule platPreuViNegre
 (object (is-a Plat) (vi ?v))
 (object (name ?v) (tipus negre))
 =>
 (print t " preu " (send ?v get-Preu) crlf)
)
```

Aquesta regla imprimiria el preu del vi asociat al plat si es negre.

3.5.5 Tengo problemas en condiciones de reglas con slots de instancias obtenidos con `send`

A veces dan problemas las reglas en las que las condiciones sobre los valores de los slots de una instancia se escriben usando mensajes get sobre los atributos de una instancia en lugar de utilizar patrones para obtenerlos, por ejemplo la regla:

```
(defrule platcar
 ?plat <- (object (is-a Plat) )
  (test (> (send ?plat get-Preu) 30))
 =>
 (print t (send ?plat get-nom)    " es car."
crlf)
)
```

No funciona correctamente en CLIPS, pero la regla:

```
(defrule platcar
 ?plat <- (object (is-a Plat) (Preu ?p))
 (test (> ?p 30))
 =>
 (print t (send ?plat get-nom) " es car." crlf)
```

Sí lo hace.

En el caso de que el valor del slot sea booleano se puede escribir la regla como:

```
(defrule platpicant
 ?plat <- (object (is-a Plat) (Picant TRUE))
 =>
 (print t (send ?plat get-nom) " es picant." crlf)
```

En el caso de que el slot sea a su vez una instancia, se puede utilizar para obtener los valores de esta instancia en la condicion como se explica en la pregunta anterior.

3.5.6 Tengo problemas con instancias obtenidas del slot de otra instancia en el patrón de una regla

Este problema aparece cuando el patron de una regla instancia una variable a un nombre de instancia y esta no esta definida en el modulo de la regla, por ejemplo, si la regla siguiente pertenece a un modulo que no es el de la instancia:

```
(defrule platPreuViNegre
 (object (is-a Plat) (vi ?v))
 (object (name ?v) (tipus negre))
 =>
 (print t " preu " (send ?v get-Preu) crlf)
)
```

La regla se quejará de que la instancia no existe al enviarle el send ya que solo se busca la instancia dentro del módulo de la regla y no en los importados. Para arreglar esto se puede

usar la función `(instance-address [<MODULO>|*] <INSTANCIA>)`, que busca la direccion de la instancia en un modulo concreto o en todos los módulos que se importan (`*`). Deberíamos entonces escribir la regla:

```
(defrule platPreuViNegre
 (object (is-a Plat) (vi ?v))
 (object (name ?v) (tipus negre))
 =>
 (print t " preu " (send (instance-address * ?v) get-Preu) crlf)
)
```

3.5.7 ¿Cómo realizar una búsqueda de instancias que cumplan unas restricciones?

CLIPS nos ofrece varias funciones de búsqueda de instancias que cumplan unas determinadas restricciones que a nosotros nos interese:

- La función `find-instance`: Esta función devuelve la primera instancia que cumple todas las restricciones indicadas.
- La función `find-all-instances`: Esta función devuelve todas las instancias que cumplen las restricciones indicadas.

La sintaxis de ambas funciones es igual exceptuando el nombre de la función:
`(find-all-instances (clase_instancias) (restricciones))`.
A continuación se muestran varios ejemplos:

- `(bind ?rectangulos (find-all-instances ((?inst Rectangulo)) (> ?inst:altura 10)))`. Después de ejecutar esta función, en la variable rectangulos habrá una lista con todas las instancias de la clase Rectangulo cuya altura sea superior a 10.
- `(bind ?rectangulos (find-all-instances ((?inst Rectangulo)) (and (> ?inst:altura 10) (= ?inst:anchura 7))))`. Este trozo de código hace que en la variable rectangulos se guarde una lista de las instancias de la clase Rectangulo que tienen una altura superior a 10 y una anchura igual a 7.

Estas funciones que CLIPS nos ofrece tienen otra funcionalidad y es que podemos realizar búsquedas de conjuntos de instancias:

- `(bind ?rectangulos (find-all-instances ((?a Rectangulo) (?b Rectangulo)) (= ?a:altura ?b:altura)))`. Con este trozo de código conseguiríamos obtener todas las parejas de instancias de la clase Rectangulo que tienen la misma altura.

Por último, es necesario comentar que se pueden realizar búsquedas de conjuntos de instancias que sean de distintas clases:

- `(bind ?figuras (find-all-instances ((?rect Rectangulo) (?circ Circulo)) (= ?rect:altura ?circ:radio)))`. Con este trozo de código conseguiríamos obtener todas las parejas `<rectangulo,circul` que cumplan que la altura del rectángulo sea igual al radio del círculo.

Existen más comandos para obtener instancias, pero son menos utilizadas a nivel básico, como pueden ser, `any-instancep`, `do-for-instance`, `do-for-all-instances` y `delayed-do-for-all-instances`.

3.5.8 ¿Cómo se interactúa con objetos?

La interacción con objetos se efectúa mediante lo que se denomina mensaje.

```
(defmessage-handler
  <nombre-clase>
  <nombre-mensaje>
  [<tipo-handler>] [<comentario>]
  <parámetro>* [<parámetro-comodín>])
  <acción>*
```

Un gestor de mensajes consta de 7 partes:

1. Nombre de clase a la que el gestor estará asociado
2. Nombre del mensaje
3. Tipo de gestor (Nosotros habitualmente usaremos primary que viene por defecto)
4. Comentario (opcional);
5. Lista de parámetros
6. Parámetro comodín (para gestionar múltiples parámetros)
7. Secuencia de acciones o expresiones que serán ejecutadas por el gestor

Enfocado a la práctica de CLIPS, la más común es que queramos interactuar con objetos para:

1. Imprimir los objetos Un ejemplo:

```
1     (defmessage-handler avion imprimir-beneficio ()
```

2. Para implementar funciones calculadas asociadas al objeto Un ejemplo:

```
1   (defmessage-handler avion calcular-beneficio ()
2    (* ?self:plazas-ocupadas ?self:precio-billete))
3   )
```

Como se puede observar el parámetro implícito `?self`, contiene la instancia activa para este mensaje.

3.5.9 ¿Como se interactúa con las instancias de los objetos?

Podremos interactuar entre las instancias de los objetos mediante la función `send`.

```
(send <expresión-de-objeto> <nombre-de-mensaje> <expresión>*)
```

Donde se toman como argumentos el objeto destino del mensaje, el mensaje mismo, y otros parámetros que debieran ser pasados a los gestores.

Particularmente, en el ámbito de la práctica, usualmente necesitaremos utilizar el envió de mensajes de tipo `get`, `put` and `delete`. Estos mensajes tiene la siguiente sintaxis:

- `get-<nombre-atributo>`

- put-<nombre-atributo>
- delete

Aquí tenemos un ejemplo de como utilizarlo, recordando que ponemos entre corchetes la instancia la cual se envía el mensaje

```
(defclass avion (is-a USER)
     (role concrete)
     (slot precio-billete (create-accessor read) (default 34))
     (slot plazas-ocupadas (create-accessor write) (default 0)))
;Clase creada

>(make-instance a of avion)          ;Creación de una instancia de avion
[a]
>(send [a] get-precio-billete)       ;Obtención de un slot
34                                   ;Resultado obtenido
>(send [a] put-plazas-ocupadas 100) ; Modificación de un slot
>(send [a] delete)                  ; Eliminación de la instancia
True
```

3.5.10 Uso de make-instance

Cuando queremos crear una instancia de una clase para ir rellenandola con los resultados o datos que vayamos obteniendo se haria lo siguiente:

```
(bind ?variable_instancia (make-instance nombre_instancia of nombre_clase))
```

Una vez creada, podemos jugar con dicha instancia en las diferentes reglas, siempre que la llamemos en los activadores:

```
(defrule regla
 ?var_instancia <- (object (is-a nombre_clase))
 =>
 ...
```

De esta manera dentro de la regla podemos hacer acciones con la instancia, por ejemplo:

```
(send ?var_instancia put-articulo ?articulo_ejemplo)
```

3.5.11 Como generar automáticamente nombres de instancia

En ocasiones queremos crear instancias en reglas y queremos que su nombre sea distinto del de otras que ya tenemos. Para ello podemos utilizar las funciones `gensym` y `gensym*`. La primera genera un símbolo del estilo `genN` donde N es un número. La segunda hace lo mismo, pero asegurándose de que ese símbolo no exista ya. Se puede reiniciar el contador que usan estas dos funciones con el operador `(setgen <num>)` donde `<num>` es el número por el que queremos que empiecen ahora los símbolos.

Para usar esto para generar el nombre de una instancia podemos hacer:

```
(make-instance (gensym) of <clase>)
```

Cada vez que se ejecute se generará un nombre de instancia nuevo.

Si creamos instancias de diferentes clases y queremos que las instancias tengan nombres que podamos identificar podemos concatenar símbolos a los símbolos generados por `gensym`, por ejemplo:

```
(make-instance (sym-cat pepe- (gensym)) of <clase>)
```

generará una instancia con el nombre `pepe-genN`.

3.5.12 Com fer un get/set d'un slot d'una classe?

Per consultar el valor d'un slot d'una classe, ho podem fer enviant-li un missatge de la següent manera:

```
(send <VARIABLE> get-<NOM_ATRIBUT>)
```

Per exemple, si tenim una instància de la classe Plat (`?instancia`) i volem saber el valor del seu slot nom:

```
(send ?instancia get-nom)
```

Per assignar un valor a un slot d'una d'instància, hem d'enviar-li un missatge de la següent manera:

```
(send <VARIABLE> put-<NOM_ATRIBUT> <VALOR>)
```

On `<VALOR>` pot ser un valor o un mètode que retorni un valor.

```
(send ?instancia put-nom "nom")
```

3.5.13 ¿Cómo acceder a una instancia cuyo nombre conocemos?

Si conocemos el nombre de una instancia y necesitamos acceder a ella, es decir, necesitamos enviarle algún mensaje mediante la función send, podemos hacerlo directamente escribiendo, en lugar de la variable que almacena la instancia, el nombre de la instancia entre corchetes `[ ]`. A continuación se muestra un ejemplo:

```
(bind ?altura_rect1 (send [rect1] get-altura))
```

En este caso, se guardará en la variable `altura_rect1` el valor de la altura de la instancia que tiene como nombre `rect1`.

3.5.14 Obtenir el nom d'una classe

Si tenim una instància d'una classe i volem saber el nom de la classe (útil si tenim una bona taxonomia de classes) ho podem fer amb el mètode `(class <INSTANCIA>)`.

Si per exemple, imaginem que tenim la classe Actor, la qual té com subclasses la classe APrincipal, ASecundari i ACameo. Imaginem que estem tractant amb instancies d'actors, i volem saber si la instancia que estem tractant es de la classe APrincipal.

Ho hauríem de fer de la següent manera:

```
(eq (class ?instancia) APrincipal)
```

Fixem-nos que `APrincipal` és un símbol, no un String (un truco és fixar-se en que no té cometes). Si volguéssim comparar-ho amb un String (per exemple, un String introduït per l'usuari) ho podríem transformar a un String amb el mètode `str-cat`.

```
(eq (str-cat (class ?instancia)) "Principal")
```

3.5.15 Obtenir els noms de les superclasses d'una classe

Suposem que tenim una instància de la qual volem saber-ne les seves superclasses. Suposem la instància ?inst, que és de la classe C, que té com a superclasse B que té, per la seva banda, una superclasse que es diu A. Per obtenir la llista de superclasses de la instància ?inst podem fer servir la instrucció class-superclasses. La seva sintaxi és:

```
(class-superclasses <class-name> [inherit])
```

Aquesta funció es pot fer servir de dues maneres: amb la opció inherit, que retorna tota la branca de classes que la classe de nom <class-name> té per sobre, o sense:

- Sense la opció inherit

  ```
  (bind ?superClasses (class-superclasses (class ?inst)))
  ```

  Una vegada s'executi la funció, la variable ?superClasses tindrà tots els noms de les superclasses de la variable ?inst. En l'exemple: ?superClasses = B. En cas que la classe C heredés d'una altra classe D, ?superClasses = B D.

- Amb la opció inherit

  ```
  (bind ?superClasses (class-superclasses (class ?inst) inherit))
  ```

  Una vegada s'executi la funció, la variable ?superClasses tindrà tots els noms de les superclasses de la variable ?inst i també els noms de les superclasses de les seves superclasses, i així fins a arribar a la classe OBJECT. En l'exemple: ?superClasses = B A USER OBJECT. En cas que la classe C heredés d'una altra classe D: ?superClasses = B D A USER OBJECT.

  Cal dir que aquesta funció retorna una llista de stings sense repeticions.

3.5.16 Obtenir els noms de les subclasses d'una classe

Suposem que tenim una instància de la qual volem saber-ne les seves subclasses. Suposem la instància ?inst que és de la classe A, que té com a subclasse B que té, per la seva banda, una subclasse que es diu C. Per obtenir la llista de subclasses de la instància ?inst podem fer servir la instrucció class-subclasses. La seva sintaxi és:

```
(class-subclasses <class-name> [inherit])
```

Aquesta funció es pot fer servir de dues maneres: amb la opció inherit, que retorna tota la branca de classes que la classe de nom <class-name> que té per sota, o sense:

- Sense la opció inherit

  ```
  (bind ?subClasses (class-subclasses (class ?inst)))
  ```

  Una vegada s'executi la funció, la variable ?subClasses tindrà tots els noms de les subclasses de la variable ?inst. En l'exemple: ?subClasses = B. En cas que la classe A tingués una altra subclasse D, ? subClasses = B D.

- Amb la opció inherit:

  ```
  (bind ?subClasses (class-subclasses (class ?inst) inherit))
  ```

  Una vegada s'executi la funció, la variable ?subClasses tindrà tots els noms de les subclasses de la variable ?inst i també els noms de les subclasses de les seves subclasses, fins a arribar

a les classes que estiguin a les fulles. En l'exemple: ?subClasses = B C. En cas que la classe A tingués una altra subclasse D, ?subClasses = B D C.

Cal dir que aquesta funció retorna una llista de stings sense repeticions.

3.5.17 Como convierto un INSTANCE-ADRESS en un INSTANCE-NAME?

En algún momento nos podemos encontrar con la situación de que disponemos de un puntero a una instancia y necesitamos su nombre o lo contrario. En ambos casos podemos usar la función instance-name. Su funcionamiento es bidireccional, podemos introducirle un instance-name y nos devolverá un instance-adress y viceversa.

```
1 (defrule imprimir-nombre
2  ?est<-(object (is-a Estudiante) (edad 25))
3  =>
4  (printout t (instance-name ?est) crlf)
5 )
```

3.5.18 Com iterar a través d'un multi-slot?

El següent també és vàlid com exemple de com iterar a través d'una llista.

Imaginem que tenim una instància de la classe Plat a la varible (`?plat`), la qual té un multi-slot d'instàncies de la classe Ingredient. Imaginem que volem imprimir per pantalla el nom d'aquests ingredients (slot nom de la classe Ingredient): Necessitarem iterar sobre la llista d'ingredients.

```
1 (bind ?i 1)
2  (while (<= ?i (length$ (send ?plat get-ingredients)))
3   do
4   (bind ?ingredient (nth$ ?i (send ?plat get-ingredients)))
5   (printout t (send ?ingredient get-nom) crlf)
6  (bind ?i (+ ?i 1))
7 )
```

El mètode `(nth$ <INDEX> <LLISTA>)` et retorna l'element situat a `INDEX` de `LLISTA`.

El mètode `(length$ <LLISTA>)` retorna el número d'elements de LLISTA.

3.5.19 Cómo modificar/insertar/borrar valores en un multislot

Los valores de un multislot se pueden cambiar usando la función `slot-replace$`. La sintaxis es la siguiente:

```
(slot-replace$ <nom-instancia> <nom-multislot> <inicio> <fin> <valor>*)
```

Donde `<inicio>` y `<fin>` son el rango de posiciones del multislot que se quiere modificar y `<valor>*` son los valores que queremos reemplazar. Obviamente deberá haber tantos valores como posiciones hayamos indicado en el rango.

Para insertar nuevos valores podemos usar la función `slot-insert$`. La sintaxis es la siguiente:

```
(slot-insert$ <nom-instancia> <nom-multislot> <pos> <valor>)
```

El valor se inserta delante de la posición indicada, si la posición es mayor que la longitud del multislot se colocará al final.

Para borrar elementos de un multislot podemos usar la función `slot-delete$`. La sintaxis es la siguiente:

```
(slot-delete$ <nom-instancia> <nom-multislot> <inicio> <fin>)
```

Se borrarán los elementos que están en el rango indicado.

3.5.20 Cómo recorrer un atributo multievaluado

Supongamos que la variable respuesta tiene una lista de instancias

```
1 (loop-for-count (?i 1 (length$ ?respuesta)) do
2  (bind ?aux (nth$ ?i ?respuesta))
3  ;aquí hacemos lo que queramos
4  )
5 )
```

`Loop-for-count` funciona como un for, primero la variable que hará de contador, luego el valor de origen y luego el valor final. Para controlar el final usamos la función que he explicado antes. `i` tomará valores desde 1 hasta n, así que usando la función `nth$` obtendremos cada uno de los elementos. La variable `?aux` irá teniendo cada uno de los valores.

3.5.21 Cómo recorrer un atributo multievaluado de un atributo multievaluado (una matriz).

Es igual que antes pero con dos bucles

```
1 (loop-for-count (?i 1 (length$ ?respuesta)) do
2  (bind ?aux (nth$ ?i ?respuesta))
3  (bind ?aux2 (send ?aux get-slot_que_queremos))
4  (loop-for-count (?j 1 (length$ ?aux2)) do
5   (bind ?aux_final (nth$ ?j ?aux2))
6   ;aquí ya podemos trabajar con el elemento en concreto
7  )
8 )
```

Supongamos que respuesta tiene una lista de instancias, con el primer bucle haremos lo mismo que antes, ir iterando por esas instancias. Una vez que tenemos cada instancia con `aux2`, con el segundo bucle iteraremos dentro de sus instancias interiores.

Por ejemplo, si tenemos varios equipos de fútbol y cada equipo tiene varios jugadores, en el primer bucle estamos iterando entre los equipos y en el segundo entre los jugadores de cada equipo en particular.

3.5.22 Com puc esborrar una instància?

Per esborrar una instància (per exemple, segons la informació que proporcioni l'usuari podem deduir que una sèrie d'instàncies és impossible que es facin servir per a la solució) ho podem fer enviant un missatge delete a la instància, és a dir `(send <INSTANCIA> delete)`.

```
(send ?inst delete)
```

Un mètode que podria ser útil de la cara a la pràctica podria ser el següent:

```
(defrule elimina-instancia
 (declare (salience 10))
 ?elimina-fact <- (elimina-inst ?inst)
 =>
 (send ?inst delete)
 (retract ?elimina-fact)
```

Imaginem que per exemple que volem eliminar una llista d'instàncies. Per eliminar-les, podríem recórrer el llistat marcant les instàncies que volem esborrar afegint el següent fet:

```
(assert (elimina-instancia ?inst))
```

Així, com que elimina-instancia té un salience alt esborraria les instàncies marcades quan tingués ocasió.

D'aquesta manera ens estalviem el problema de recórrer un llistat d'instàncies amb un iterador mentre anem eliminant els elements de la llista, cosa que pot donar problemes amb l'iterador.

3.5.23 Com imprimir una instància?

Per imprimir una classe (útil per exemple per mostrar el resultat final) el millor és definir message-handlers.

Per exemple, imaginem que tenim una classe `Plat`, la qual té un slot `nom` de tipus String. Si volguéssim imprimir per pantalla un missatge que digués:

```
El nom del plat és --<nom del plat>--
```

Hauríem de definir un message-handler que simplement imprimís per pantalla el valor de l'slot, de la forma:

```
(defmesssage-handler <CLASSNAME> <MESSAGENAME> primary ()
   <CODI>
)
```

Per el cas de l'exemple:

```
(defmessage-handler Plat print primary ()
 (printout t "--" ?self:nom "--") )
)
```

Per cridar-ho, només caldrà enviar-li un missatge a la instància amb el nom del message-handler (en aquest cas, `print`).

```
(send ?instancia print)
```

Un message-handler pot cridar a altres message-handlers, amb el que es poden encadenar diverses instàncies. Per exemple, imaginem que tenim la classe Menú, la qual té un atribut plats que és un llista d'instàncies de la classe Plat. Dintre del message-handler menú, podríem iterar a través de la llista de plats enviant el missatge print que hem definit anteriorment.

3.5.24 Com es navega entre instàncies relacionades?

Per exemple, tinc una instància iA que té un slot anomenat slotB que és una instància d'un frame B. Vull saber el nom de la instància associada al slotB de iA. Ho pots fer així:

```
(bind ?B (send ?iA get-slotB))
(bind ?res (send ?B get-nom))
```

Si estàs dins d'una condició, ho pots simplificar de la forma següent:

```
(bind ?res (send ?iA:slotB get-nom)
```

Els dos punts són equivalents al punt de Java

3.6 Programación

3.6.1 Com introduir un comentari?

Els comentaris a CLIPS va darrere de un `;`, com per exemple:

```
;Comentari
```

També podem afegir comentaris a les nostres regles, deffacts, deftemplates, etc. Generalment van darrere el nom de la estructura, per exemple:

```
1 (defrule neteja "regla per la qual si la cuina està bruta s'afegirà una tasca de
2 neteja"
3  (estat-cuina bruta)
4   =>
5  (assert (tasca netejar))
6 )
```

3.6.2 Referenciar el valor null a CLIPS?

Per referenciar el valor null a clips es fa amb el simbol `nil`. Per exemple, si volem saber si una instància `?inst` es null, ho podríem fer de la següent manera:

```
(eq ?inst nil)
```

3.6.3 Formas de recorrer una lista:

Para recorrer una lista disponemos de diferentes opciones, en este FAQ explicaremos como crear una estructura tipo "WHILE" y tipo "FOR".

1. `While`: Esta regla se ejecuta hasta que la expresión a evaluar sea falsa. Este tipo de estructura se rige por seguir la sintaxis `while - do`, y se construiría de la siguiente forma:

   `(while <expresión> [do] <acción>)`

   Dónde dependiendo del valor de `<expresión>`:

   - Valor TRUE: Se realizará la acción `<acción>` i se volverá a evaluar `<expresión>`.
   - Valor FALSE: No se ejecutará la acción `<acción>`

2. `For`: Esta regla sirve para ejecutar un bucle N (final - inicio) veces. Este tipo de estructura se rige por seguir la sintaxis `loop-for-count`, , y se construiría de la siguiente forma:

   `(loop-for-count (<var> <inicio> <final>) [do] <acción>)`

   Dónde:

   - `<var>`: Variable que se usara como índice.

- `<inicio>`: Valor inicial que tendrá `<var>`
- `<final>`: Ultimo valor que tendrá la variable `<var>`
- `<acción>`: Acción que es realizará hasta que `<var>` tenga el valor de `<fin>`

3.6.4 Formas de elegir una opción

En este apartado explicaremos como crear una estructura `if` y una estructura `switch`.

- `If`: Este tipo de estructura se rige por seguir la sintaxis `if - then - else`, y se construiría de la siguiente forma:

  `(if <expresión> then <acción> [else <acción2>])`

  Dónde dependiendo del valor de <expresión>:

  - Valor TRUE: Se realizará la acción `<acción>`.
  - Valor FALSE: Se realizará la acción `<acción2>` (en caso de haber)

- `Switch`: Este tipo de estructura se rige por seguir la sintaxis `switch - opciones`, y se construiría de la siguiente forma:

```
(switch   <expresión-test>
 (case OPCION1   then   ACCION1)
 (case OPCION2   then   ACCION2)
             .
             .
             .
 ([default]   ACCION-DEFAULT)
)
```

  Dónde dependiendo del valor de `<expresión-case>` se ejecutará una `<acción>` en particular. Si ninguna `<opción>` es correcta, se ejecutara la `<acción-default>`.

3.6.5 Funciones con la clase String

Habitualmente trabajaremos con Strings o cadena de Strings. CLIPS proporciona una serie de herramientas para trabajar con esta clase:

1. Convertir string a MAYÚSCULAS: A veces es conveniente tener todos los caracteres en mayúsculas para no tener problemas de comparaciones o simplemente por convenio. Para ello, CLIPS dispone de la función `upcase`, la cual convierte a mayúsculas un símbolo o string.

   `(upcase <expresión-simbólica-o-de-cadena> )`

   Ejemplos:

   `(upcase "Clips es divertido")` → `"CLIPS ES DIVERTIDO"`

   `(upcase Clips_es_divertido)` → `CLIPS_ES_DIVERTIDO`

2. Convertir string a MINÚSCULAS: la función es `downcase`.

3. Concatenación de strings: También es de gran utilidad poder juntar dos o más parámetros en un solo string. Todo parámetro a juntar deben ser de uno de los siguientes tipos: symbol. String, integer, float o instance-name.

   `(str-cat parametro1 parametro2 ... parametroN )`

Ejemplo:

`(str-cat "IA" nota 10)` → `"IAnota10"`

4. Comparación de strings: Función que nos indica si dos strings son iguales.

```
(str-compare
      <expresión-simbólica-o-de-cadena>
      <expresión-simbólica-o-de-cadena>
```

Esta función devuelve:

0: Si ambos strings son iguales.

1: Si el primer string es mayor que el segundo string.

-1: Si el primer string es menor que el segundo string.

5. Longitud de un string: Muchas veces es de gran utilidad saber que tamaño tiene un string.:

`(str-length <expresión-simbólica-o-de-cadena>)`

Ejemplo:

`(str-length "Inteligencia Artificial")` → 23

3.6.6 ¿Cómo obtener el contenido de una variable en un string?

Para conseguir un string a partir de un integer, float, symbol, instance-name, ... debes utilizar el comando `str-cat`. Si a la función le pasamos como parámetros varios elementos, los transformará a string y los concatenará. La sintaxis de esta función es la siguiente: `(str-cat elemento/s)`. Esta función la usaremos sobre todo cuando queramos realizar comparaciones entre el contenido de una variable y un string ya que previamente, necesitaremos usar la función `str-cat` para pasar el contenido de la variable a string y realizar de este modo la comparación de forma correcta.

3.6.7 ¿Como se crea una lista?

Con `create$`. Todos los operadores sobre listas llevan el símbolo $ al final. Igual que con las variables se debe usar bind siempre que se desee modificar una lista.

```
(bind ?lista (create$ a b c))
```

3.6.8 ¿Como borro un elemento de una lista?

Con `delete$`.Usa la siguiente sintaxis:

`(delete$ <lista> <indice-inicio> <indice-final>)`.

Se borrarán todos los elementos del rango entre las posiciones inicio y final, ambas incluidas. También existe

`delete-member$`: `(delete-member$ <lista>  <elemento>)`

Borrará todas las apariciones del elemento en la lista.

`(delete$ (create$ a b c d e f) 3 5)` ⟶ `(a b f)`

3.6.9 ¿Como inserto elementos en una lista?

Con `insert$`. Sintaxis:

`(insert$ <lista> <indice> <expresión simple o lista>)`.

Inserta todos los valores simples o de multicampo en la lista antes del índice-ésimo valor (`<índice>` debe ser un entero) de la lista dada.

`(insert$ (create$ a b c d) 1 x)` ⟶ `(x a b c d)`

3.6.10 ¿Como modifico elementos de una lista?

Con `replace$`. Sintaxis:

`(replace$ <lista-a-modifcar> <inicio> <final> <lista-nuevos-valores>)`.

Los valores entre las posiciones inicio y final serán substituidos por la nueva lista. Recordar que el uso de bind es imprescindible si queremos conservar los cambios.

`(replace$ (create$ a b c) 3 3 x)` ⟶ `(a b x)`

3.6.11 ¿Como averiguo el número de elementos de una lista?

Con `length$`. Sintaxis `(length$ <lista>)`.

3.6.12 ¿Como consulto un valor de la lista a través de su posición?

A través de `nth$`. Sintaxis: `(nth$ <índice> <lista>)`.

3.6.13 ¿Como recorro todos los elementos de una lista?

Manualmente con un bucle y `nth$`, o con `progn$`. Sintaxis:`(progn$ (<var> <lista>))`. La variable iterará sobre todos los valores de la lista ordenadamente.

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

3.6.14 Com trobar si un element forma part d'una llista?

Per comprovar que un element formi part de la llista (seguint l'exemple anterior, com saber si un ingredient forma part de la llista d'ingredients) podem fer servir `(member$ <ELEMENT> <LLISTA>)`, el qual retorna un booleà indicant si l'element pertany a la llista o no.

3.6.15 Tinc un conjunt de símbols, però estan tots junts en un string. ¿Com ho faig per separar-los i posar-los en un multislot?

Has d'utilitzar la funció explode$ de CLIPS, amb la sintaxi següent:

`(explode$ string)`

Per exemple, si tenim una data en un string i volem extreure el dia:

`(bind ?dia (nth$ 1 (explode$ ?data)))`

3.6.16 I si tinc un multislot i el vull transformar en un string?

Has d'utilitzar la funció implode$ de CLIPS, amb la sintaxi següent:

`(implode$ multislot)`

Per exemple, si volem crear un string que conté una data:

```
(bind ?data (implode$ (create$ 12 1 2008)))
```

3.6.17 ¿Cómo obtener un valor aleatorio?

Es probable que en algunos casos necesitemos obtener un valor aleatorio para realizar algún tipo de acción. CLIPS nos ofrece esta posibilidad mediante la función random, que retorna un número entero aleatorio:

```
(bind ?rand (random))
```

3.6.18 Comparaciones

Para comparar dos elementos: `(operador elemento1 elemento2)`. Siendo operador `eq`, `<`, `=`, `>`, ... Por ejemplo:

```
(eq ?nuevo TRUE)
(< ?precio ?saldo)
```

Un caso específico es la igualdad ya que varía en función del tipo de los elementos comparados. Si se trata de un número se utilizará el operador `=`, en caso de que los elementos sean objetos o símbolos `eq`. Finalmente si estamos tratando cadenas debemos utilizar la función `str-compare`. Esta devuelve un número entero que representa el resultado de la comparación: 0 (son idénticos), 1 (elemento1 > elemento2) o -1 (elemento1 < elemento2).

De esta forma, si por ejemplo queremos comparar si dos cadenas son iguales, la condición sería:

```
(= (str-compare "verano" "verano") 0)
```

3.7 Los módulos

3.7.1 Què és un mòdul en CLIPS?

Els mòduls en clips són el mecanisme que ens proporciona aquest per tal de dividir, organitzar i, en alguns casos, fer més eficient, la execució de les nostres bases de coneixement. Ens poden ser útils, per exemple, per dividir un problema en diversos subproblemes, com per exemple, per dividir un problema en un primer subproblema d'anàlisis i un segon de construcció de la solució.

Els mòduls es defineixen de la següent manera:

```
(defmodule <nom-mòdul>)
```

Un mòdul només pot ser definit un cop, i no podrà ser tornat a definir mai, a excepció del mòdul `MAIN`, que es pot redefinir tan sols un cop. Cal tenir en compte que cada mòdul té la seva pròpia agenda.

3.7.2 Partición en modulos

Para que el sistema experto sea mucho más sencillo de actualizar y de comprender, es muy aconsejable hacer una partición en módulos. Lo que hacemos con esto es agrupar las reglas que busquen un objetivo en común. La construcción de un módulo se hace poniendo `(defmodule nombre_del_modulo "La descripción del módulo" (import ...) (export ...))`. Un módulo termina en donde empieza otro módulo, así que no hay que cerrar declaración, sino que se

hace implicitamente. En los apartados de import deberemos de poner todos aquellos módulos de los cuales queramos obtener y usar sus reglas. Los export son para indicar a los demás módulos que se puede usar de si mismo. Definir los módulos si luego no se puede navegar por ellos no sirve para nada. Para eso está el comando focus, el cual, seguido de un nombre de módulo, sirve para especificar que el sistema va a estar centrado, focalizado, en el módulo especificado. El uso de focus es muy adecuado como consecuente de una regla, por ejemplo:

```
1 (defrule cambio-modulo-estilo-menu
2   (Evento nombre_reserva ok)
3   (Evento numero_comensales ok)
4   (Evento temporada ?)
5   (Evento presupuesto ok)
6  =>
7   (focus preguntas-estilo-menu)
8 )
```

Cuando una regla no está en ningún módulo, es decir, el usuario no la ha incluido en ningun módulo, decimos que está en el módulo MAIN.

3.7.3 ¿Cómo asignar una construcción en un módulo?

Existen dos formas para incluir una construcción en un módulo:

Explícita: se escribe el nombre del módulo (que es un símbolo) seguido de dos puntos(`::`), que representan el símbolo separador de módulos, y a continuación el nombre. Ej. (`MÓDULO::construcción`)

Implícita: sin especificar el nombre del módulo ni los dos puntos `::`, ya que siempre existe un módulo "actual". El módulo actual cambia siempre que:

- se defina una construcción defmodule
- se especifica el nombre de un módulo en una construcción (usando `::`)
- se utilice la función `set-current-module <nombre-módulo>`.

3.7.4 Com funcionen les clàusules `export` i `import`?

Abans de res, cal advertir que només es poden exportar i importar: `deftemplate`, `defclass`, `defglobal`, `deffunction` i `defgeneric`.

Hi ha 3 maneres d'exportar o importar construccions:

1. Exportant/important el conjunt global de totes les construccions d'un mòdul

```
(defmodule modulA (export ?ALL) )
(defmodule modulB (import modulA ?ALL) )
```

2. Exportant/important totes les construccions d'un mòdul que són d'un tipus particular

```
(defmodule modulA (export deftemplate ?ALL) )
(defmodule modulB (import modulA deftemplate ?ALL) )
```

3. Exportant/important construccions específiques

```
(defmodule modulA (export deffunction funcio-util-1 funcio-util-2) )
(defmodule modulB (import modulA deffunction funcio-util-1 funcio-util-2) )
```

Per poder importar construccions d'un altre mòdul, és un requisit que aquest mòdul estigui definit abans del mòdul on estem definint la importació i que el mòdul estigui exportant les consrtuccions que voldrem usar en el nou mòdul.

Exemple:

```
1  (defmodule modulA (export deffunction funcioAdeu))
2  ; exportem només la funció funcioAdeu
3
4  (deffunction modulA::funcioAdeu ()
5   (printout t "Bye from ModuleA!" crlf)
6  )
7
8  (defmodule MAIN (import modulA ?ALL))
9  ;importem totes les construccions de modulA
10
11 (defrule MAIN::inicio
12       (initial-fact)
13       =>
14       (printout t "Hello from MAIN module!" crlf)
15       (funcioAdeu)
16 )
```

Si executem aquest codi obtindrem la següent sortida:

```
CLIPS> (run)
Hello from MAIN module!
Bye from ModuleA!
CLIPS>
```

3.7.5 ¿Cómo cambio el módulo actual?

El módulo `MAIN` es definido automáticamente por CLIPS y es el módulo actual por defecto cuando se inicia por primera vez o después de un comando clear. Cada módulo tiene su propia agenda (conjunto conflicto). Entonces la ejecución puede controlarse seleccionando una agenda, y en ésta se elegirán reglas para ejecutar. Se puede cambiar el módulo actual mediante el comando focus:

Sintaxis: `(focus <nombre-módulo>+)`

CLIPS mantiene una pila de focos, y cada vez que se hace un cambio de módulo se añade el foco actual en el top de la pila. La ejecución de las reglas continúa hasta que cambia el foco a otro módulo, no haya reglas en la agenda, ó se ejecute return en la RHS de una regla.

En las reglas existe la propiedad auto-focus, que permite ejecutar automáticamente un comando focus cuando la regla se activa:

Sintaxis: `(auto-focus TRUE | FALSE)`

3.7.6 ¿Como debería utilizar las focos en la practica de CLIPS?

La manera más natural de organizar los módulos en la práctica de clips es de manera lineal y con un orden de ejecución consecutiva. Esto quiere decir, que si nuestra práctica tiene 5 módulos estarán escritos de manera lineal m1, m2, m3, m4 y m5 y querremos que se ejecuten de manera

consecutiva : Primero m1, después m2 así consecutivamente hasta llegar a m5. De esto se deduce que al final de cada módulo, colocaremos un foco que nos redirija al siguiente módulo.

La redirección o salto al siguiente módulo puede ser de dos tipos :

1. Incondicional:

   Queremos que independiente de que reglas se hayan cumplido el foco nos redireccione al siguiente modulo. En este caso será suficiente con la sentencia

   `(focus modulo-siguiente)`

2. Condicional: Lo más habitual es queramos saltar al siguiente módulo solo si se han cumplido una serie de reglas. La solución en este caso es incorporar el foco en la parte derecha de la regla:

```
(defrule saltar-siguiente-modulo
 (declare (salience 0))
 (r1 ?)
 (r2 ?)
 (r3 ?)
 (r4 ?)
 (r5 ?)
 ->
 (focus modulo-siguiente)
)
```

Como podrá apreciar el lector se ha incluido una sentencia condicional en la parte izquierda es: `(declare (salience 0))`. Mediante esta instrucción podemos asignar una prioridad en el orden de ejecución a la regla. En este caso se ha asignado prioridad 0, suponiendo que todas las reglas restantes del módulo tienen una prioridad más alta (por ejemplo 10), asegurándonos que será la última en ejecutarse.

3.8 Funciones

3.8.1 ¿Cómo se crea una función en CLIPS?

Una deffunction se compone de cinco elementos:

1. Un nombre, que debe ser un símbolo.
2. Un comentario, que es opcional.
3. Una lista de cero o más parámetros requeridos, que deben ser variables simples)
4. Un parámetro comodín opcional que sirve para manejar un número variable de argumentos.
5. Una secuencia de acciones o expresiones que serán ejecutadas en orden cuando se llame a la función.

```
(deffunction
     <nombre>                  (1)
     [<comentario>]            (2)
     (<parámetro>*             (3)
     [<parámetro-comodín>])    (4)
     <acción>*                 (5)
)
```

El valor devuelto por la función es la última acción o expresión evaluada dentro de la función. Si una deffunction no tiene acciones, devolverá el símbolo `FALSE`. Si se produce algún error mientras se ejecuta la función, cualquier otra acción de la función aún no ejecutada se abortará, y la función devolverá el símbolo `FALSE`.

Ejemplo:

```
(deffunction
   mostrar-params                                    Nombre (1)
   (?a ?b                                            Parámetro  simples (3)
   $?c)                                              Parámetro múltiple  (4)
   (printout t ?a " " ?b " and " (length ?c)
                          " extras: " ?c crlf)       Acción (5)
```

La función anterior guarda los dos primeros parámetros dentro de las variables `?a` y `?b` y el resto (en caso de haber) en `$?c`. Esta función realiza la acción de imprimir por pantalla un mensaje formado por:

Los valores de las variables `a` y `b` + and + longitud de la variable `c` + extras + valores de `c`

Si ejecutamos la función definida anteriormente con dos parámetros: `(mostrar-params 1 2)` el valor de cada variable será:

?a = 1

?b = 2

?c = Vacio

E imprimirá: `1 2 and 0 extras: ()`

Si ejecutamos la función definida anteriormente con cuatro parámetros: (mostrar-par a b c d) el valor de cada variable será:

?a = a

?b = b

?c = c d

E imprimirá: `1 2 and 2 extras: (c d)`

3.8.2 Explicación de la estructura de una función en general.

Hacer una función en CLIPS es igual que en cualquier otro lenguaje. Se ha de tener en cuenta que la función no tiene activadores como en una regla. Únicamente, ésta tiene que ser colocada dentro del módulo donde va a ser utilizada y siempre tiene que devolver algo (tal como se verá en el ejemplo de estructura siguiente). Ejemplo:

Si en pseudocódigo tuviésemos:

```
funcion devuelvo_boolean (lista L1, lista L2)
{
 boolean b = false;
 int i = 1;
 mientras(i<=N && !b)
 {
```

```
7     ...
8     i++;
9     }
10   return b;
11  }
```

En CLIPS quedaria así:

```
1  (deffunction devuelvo_boolean (?L1 ?L2)
2    (bind ?b FALSE)
3    (bind ?i 1)
4    (while (and (<= ?i  N) (not ?b))
5      do
6      ...
7      (bind ?i (+ ?i 1))
8    )
9   ?b
10 )
```

3.8.3 Com definir un paràmetre d'una funció com una llista?

Si volem que un paràmetre de la funció sigui una llista, ho haurem d'indicar amb un `$`. Per exemple,

```
(deffunction pertany (?var $?llista) (member$ ?var ?llista))
```

Retornaria si var forma part d'una llista.

La manera d'indicar que un paràmetre és una llista seria la mateixa per els fets a la part esquerre d'una regla.

3.8.4 Com puc utilitzar una funció a la part esquerra de les regles?

Fent servir la instrucció `test`. test es satisfà si la funció retorna qualsevol valor diferent de FALSE

```
(test (estaALaLlista ?persona1 $?personesConvidades ))
```

3.9 Entrada salida

3.9.1 ¿Cómo se imprime por pantalla?

Para imprimir por pantalla sin especificar el formato de cada parámetro se usa la función `printout`.

```
(printout  nombre-logico  <parámetro>+  )
```

- **Nombre-lógico:** Envía al dispositivo asociado con el nombre lógico especificado un string formateado. Puede tener uno de los siguientes valores:
  - **Nil:** No se produce salida pero devuelve el string formateado.
  - **T:** Se imprime en la salida estándar.

- **Parámetros:** Los parámetros pueden ser:
  - **Variables:** Si queremos imprimir el valor de una variable debemos escribir un interrogante y su nombre: `?nombreVariable`
  - **Resultado de funciones:** Si queremos escribir el valor de retorno de una función deberemos escribir entre paréntesis la función a ejecutar: `(nombreFuncion)`
  - **Strings:** Si queremos escribir una cadena des Strings, deberemos poner la cadena de strings entre comillas: `"loQueQuieraEscribir"`.

Si por el contrario queremos especificar el tipo de cada parámetro a imprimir, usaremos la función `format`.

```
( format <nombre-logico> <string-control> <parámetros>*)
```

- **Nombre-lógico:** Envía al dispositivo asociado con el nombre lógico especificado un string formateado. Puede tener uno de los siguientes valores:
  - **Nil:** No se produce salida pero devuelve el string formateado.
  - **T:** Se imprime en la salida estándar.
- **String-Control:** El string de control contiene unos flags de formato que indican cómo serán impresos los parámetros. Estos flags son de la forma : `%[-][M][.N]x`, donde:
  - -: Es opcional y significa justificado a la izquierda (por defecto derecha).
  - M: Indica la anchura del campo en columnas. Como mínimo se imprimirán M caracteres.
  - N: Específica el numero de dígitos a la derecha del punto decimal y es opcional. Por defecto se toman 6 para los números reales.
  - X: Especifica el formato de impresión y puede ser:
    - d: Entero
    - f: Decimal
    - e: Exponencial (potencias de 10).
    - g: General (numérico). Imprimir con el formato mas corto.
    - o: Octal. Numero sin signo. (N no se aplica)
    - x: Hexadecimal.
    - S: String.
    - N: Salto de línea.
    - R: Retorno de carro.
    - %: el carácter "%".
- `Parámetros:` Parámetros a imprimir.

3.9.2 ¿Al imprimir por pantalla como hago un salto de línea?

Con `crlf`.

3.9.3 Com imprimeixo una línia en blanc?

Sense escriure cap text:

```
(printout t crlf)
```

3.9.4 Com llegir de la entrada standard?

Per llegir de l'entrada estàndard CLIPS ofereix el mètode `(read)`. Generalment, per llegir el que vol entrar un usuari farem:

```
(bind ?text (read))
```

Això deixarà a la variable `?text` el que escrigui l'usuari fins que faci un retorn de carro.

Si volem comprovar que la resposta sigui un String podem fer servir, `(lexemep <VALOR>)` el qual comprova que `VALOR` sigui un String o un Symbol.

Si el que volem és un enter, podem fer servir `(integerp <VALOR>)`, que comprova que VALOR sigui un enter.

3.9.5 Vull llegir un número (o conjunt de números) com un string, però CLIPS me'ls llegeix com un enter

Utilitza la funció `(readline)` en lloc de la funció `(read)`.

nota adicional: Las funciones `(read)` y `(readline)` tienen un comportamiento diferente. La función `(read)` espera un elemento válido de CLIPS en la entrada (un entero, un real, un string, un símbolo, una instancia, incluso una lista) e ignora todo lo que queda en la entrada una vez ha reconocido el elemento válido. Este comportamiento es habitual en muchos lenguajes de programación no imperativos (funcionales y declarativos) como por ejemplo LISP o Prolog donde los procedimientos de lectura no se limitan a reconocer caracteres, sino que son conscientes de las estructuras del lenguaje. La función `(readline)` es la función de tratamiento de lectura de caracteres habitual en los lenguajes imperativos.

3.9.6 Haig de fer una pregunta a l'usuari, i la resposta és un conjunt d'elements (no sé d'entrada quants), com ho faig per assignar-los a un multislot?

Per fer això primer necessites definir una funció que llegeixi un string d'entrada (que contindrà el conjunt d'elements) i després trencar la cadena per poder distingir cada element individualment ("pasta marisc fruita" $->$ "pasta" "marisc" "fruita"). A l'exemple següent es veu com fer-ho:

```
1  ; Fa una pregunta sobre una llista d'elements
2  (deffunction pregunta-llista (?pregunta)
3    (format t "¿%s?" ?pregunta)
4    ; Llegim una línea sencera (Ex. "Pasta Marisc Fruita")
5    (bind ?resposta (readline))
6    ; Separem l'string (Ex. "Pasta" "Marisc" "Fruita")
7    (bind ?res (explode$ ?resposta))
8    ; Retornem els diferents camps (Ex. "Pasta" "Marisc" "Fruita")
9    ?res
```

Un cop tenim la funció, l'únic que hem de fer és assignar els valors que hem llegit i processat a la variable que ens interessi. Per exemple, suposem que volem assignar la resposta a una variable anomenada ingredients:

```
(bind ?ingredients
  (pregunta-llista "Hi ha algun ingredient que no desitgi incloure al menu")
)
```

3.10 Funciones útiles

3.10.1 Obtener una respuesta de un conjunto predefinido de respuestas posibles

```
(deffunction   pregunta  (?pregunta  $?valores-permitidos)
       (progn$
         (?var ?valores-permitidos)
         (lowcase ?var))
       (format t "¿%s? (%s) " ?pregunta (implode$ ?valores-permitidos))
       (bind ?respuesta (read))
       (while (not (member$ (lowcase ?respuesta) ?valores-permitidos)) do
         (format t "¿%s? (%s) " ?pregunta (implode$ ?valores-permitidos))
         (bind ?respuesta (read))
       )
    ?respuesta
)
```

Esta función guarda en el parámetro `respuesta` la respuesta elegida. La función es muy simple:

1. Convierte a minúsculas el conjunto de valores permitidos para prevenir errores de comparación.
2. Lee de teclado y guarda la respuesta en la variable `respuesta`
3. Mientras la respuesta no sea una de las permitidas sigue haciendo la pregunta.
4. Devuelve la respuesta correcta.

3.10.2 Obtener un valor numérico comprendido en un rango

```
(deffunction   pregunta-numerica (?pregunta ?rangini ?rangfi)
 (format t "¿%s? [%d, %d] " ?pregunta ?rangini ?rangfi)
 (bind ?respuesta (read))
 (while (not(and(> ?respuesta ?rangini)(< ?respuesta ?rangfi))) do
        (format t "¿%s? [%d, %d] " ?pregunta ?rangini ?rangfi)
        (bind ?respuesta (read))
 )
 ?respuesta
```

Esta función guarda en el parámetro `respuesta` el valor elegido. La función es muy simple:

1. Lee de teclado y guarda la respuesta en la variable `respuesta`
2. Mientras el valor no este contenido dentro del rango deseado sigue haciendo la pregunta.
3. Devuelve la respuesta correcta.

3.10.3 Realizar una pregunta general

```
(deffunction   pregunta-general  (?pregunta)
  (format t "¿%s?" ?pregunta)
  (bind ?respuesta (read))
  ?respuesta
```

Realiza una pregunta y almacena la respuesta en `respuesta`

3.10.4 ¿Cómo se realiza una pregunta binaria?

```
(deffunction si-o-no-p (?pregunta)
    (bind ?respuesta (pregunta ?pregunta si no s n))
    (if (or (eq (lowcase ?respuesta) si) (eq (lowcase ?respuesta) s))
        then TRUE
        else FALSE
    )
```

Devuelve cierto si se escribe `s` o `si` y falso en caso contrario.

3.10.5 Encuentra la instancia con valor mínimo para un slot

- El primer parámetro es la lista de instancia.
- El segundo parámetro es el método con el que se accede al slot.
- El tercer parámetro es el valor con que se inicializa la instancia.

Devuelve FALSE si no encuentra ninguno o si la lista está vacía.

```
(deffunction minimum-slot (?li ?sl ?init)
 (bind ?encontrado FALSE)
 (if (neq ?li FALSE) then
 (bind ?li (create$ ?li))

  (if (> (length ?li) 0) then

   (bind ?min ?init)
   (loop-for-count (?i 1 (length ?li))

    (bind ?v (send (nth$ ?i ?li) ?sl))

    (if (< ?v ?min) then

     (bind ?encontrado TRUE)
     (bind ?min ?v)
     (bind ?ins (nth$ ?i ?li))

    )
   )
  )
 )
 (if (eq ?encontrado FALSE) then
  (bind ?ins FALSE)
 )
 (return ?ins)
)
```

3.10.6 Elimina de la lista de instancias aquellas que por el multislot sl no contengan valor const

- El primer parámetro es la lista de instancia.

- El segundo parámetro es el método con el que se accede al slot.
- El tercer parámetro es el valor con que se inicializa la instancia.

Devuelve FALSE si no encuentra ninguno o si la lista esta vacía.

```
(deffunction filtrar-multi-por (?li ?sl ?const)

 (bind ?encontrado FALSE)
 (if (neq ?li FALSE) then

  (bind ?li (create$ ?li))

  (if (> (length ?li) 0) then
   (loop-for-count (?i 1 (length ?li))
    (bind $?v (send (nth$ ?i ?li) ?sl))
    (if (member$ ?const $?v) then
     (if (eq ?encontrado FALSE) then
      (bind ?encontrado TRUE)
      (bind ?ins (nth$ ?i ?li))
     else
      (bind ?ins (create$ ?ins (nth$ ?i ?li)))
     )
    )
   )
  )
 )
 (if (eq ?encontrado FALSE) then
  (bind ?ins FALSE)
 )
 (return ?ins)
)
```

3.10.7 Random slot. Devuelve una instancia aleatoria de entre las que hay en la lista li.

```
(deffunction random-slot ( ?li )
 (bind ?li (create$ ?li))
 (bind ?max (length ?li))
 (bind ?r (random 1 ?max))
 (bind ?ins  (nth$ ?r ?li))
 (return ?ins)
)
```

3.10.8 Recorre todos los elementos del slot que recibe por parámetro y los imprime por pantalla

```
(deffunction imprime-todo (?v)
  (if (> (length$ ?v) 0) then
   (loop-for-count (?i 1 (length ?v))
     (send (nth$ ?i ?v) print)
     (printout t crlf)
   )
 )
```

3.11 Ejecución de un programa CLIPS

3.11.1 ¿Cómo cargamos un programa?

Para programar en CLIPS utilizaremos ficheros con extensión CLP que contendrán el conjunto de instrucciones ha ejecutar.

3.11.2 Entorno Windows / Mac OS X

Cargar por comandos: Debemos escribir `(load "PATH/nombrefichero.clp")`

Cargar por entorno grafico: Debemos ir al menú -> load -> seleccionar el fichero.

3.11.3 Entorno Linux

Cargar por comandos: Debemos escribir (load "PATH/nombrefichero.clp")

3.11.4 ¿Cómo probar tu código en CLIPS?

Tienes que abrir la aplicación CLIPS y escribir (clear) en el Dialog Window. Posteriormente, necesitas cargar los ficheros clips. Si tienes la ontología y las reglas por separado tendras que cargar primero el fichero con la ontología y despues el de las reglas. Esto lo puedes hacer desde File->Load y seleccionando el fichero `.clp` en cuestión. A continuación debes escribir (reset) en el Dialog Window y ahora CLIPS ya estará preparado para ejecutar tu código. En concreto, lo hará cuando escribas `(run)`.

3.11.5 ¿Qué es necesario hacer entre ejecución y ejecución?

Una vez has acabado una ejecución de tu código, si por el motivo que sea quieres realizar otra, es imprescindible que escribas (reset) en el Dialog Window ya que así borras todas las reglas activadas y los hechos introducidos en la anterior ejecución. A continuación, podrías empezar la ejecución con el comando (run). Si al finalizar una ejecución, introduces algún cambio en el código y quieres probarlo, debes seguir lo explicado en la pregunta ¿Cómo probar tu código en CLIPS? pero sin tener que abrir el CLIPS de nuevo.

3.11.6 ¿Cómo parar una ejecución?

Para detener la activación de reglas se usa el comando `(halt)`. La agenda permanece intacta, y la ejecución puede reanudarse con el comando `(run)`. No devuelve ningún valor.

3.11.7 ¿Como vuelvo al estado inicial y qué contendrá éste?

Con la función `reset`. Éste contendrá todos los hechos declarados con `deffacts`, las instancias de definstances y las reglas (empezando desde cero obviamente).

4. Consejos prácticos

4.1 Tinc el disseny de la pràctica fet, però a l'hora d'implementar tot això no sé ni per on començar! Algun consell?

Bé, tothom té formes de treballar diferents, però un bon sistema pot ser el següent:

Comença amb una ontologia molt reduïda, que tingui un parell de classes i molt poques instàncies a cada classe (per poder provar coses). Després fes funcions i/o regles senzilles amb CLIPS que et permetin obtenir informació de l'ontologia que has creat. Pots començar amb regles que et retornin totes les instàncies d'una determinada classe. Segurament això al principi també et pot ser confós, aquí tens un exemple molt bàsic:

```
(defrule retorna_instancies
    (not retorna_instancies ok)
     =>
    (bind ?llista_instancies (find-all-instances ((?instancia Nom_Classe)) TRUE))
    (assert retorna_instancies ok)
)
```

Un cop tinguis una regla que funciona i que saps com funciona, es tracta d'anar-hi afegint coses a poc a poc (tant a les regles com a l'ontologia). Com ara canviant el `TRUE` de l'exemple anterior per alguna condició de l'estil

```
(eq ?instancia:nom nom_de_la_instancia_que_vull)
```

4.2 ¿Como estructuro una practica de CLIPS?

A continuación se presenta una propuesta de estructura, para facilitar el comienzo de la misma.

1. Definición de las clases

Aquí deberíamos hacer un *cortar y pegar* de todo el contenido de fichero que hemos obtenido de owl2clips a partir de la ontología en protègè en nuestro fichero de reglas.

2. Exportación del MAIN

   `(defmodule MAIN (export ?ALL))`

3. Templates

   Incluimos posibles templates que tengamos que utilizar.

4. Mensajes

   En esta parte incluiremos la comunicación (envió de mensajes) con las clases. Como habitualmente, en la práctica de CLIPS, la utilización de envío de mensajes se suele utilizar para imprimir el contenido de una clase, podríamos decir que aquí incluiremos el código de impresión de clases.

5. Funciones

   Aquí incluiremos todas las funciones que vayamos a utilizar.

6. Reglas

   La parte de reglas, como hemos comentado anteriormente estarán agrupadas en módulos. La organización de los módulos depende mucho del programador y el problema a resolver. De todas formas, existen algunos módulos que se suelen repetir como:

   (a) Módulos de preguntas.
   Será totalmente necesario obtener información sobre las preferencias y restricciones del usuario.

   (b) Módulo de selección
   Suele ser habitual hacer selección de instancias que cumplan las restricciones impuestas por el usuario.

   (c) Módulo de construcción
   Será necesario que nuestro SBC construya una solución a nuestro problema.

   (d) Módulo de impresión de resultados
   Necesitaremos un módulo final para imprimir la solución encontrada.

4.3 Com crear un flux de preguntes?

En la majoria de sistemes experts hi ha una primera fase de recopilació d'informació.

Aquesta recopilació es pot fer a través de preguntes a l'usuari. Aquestes preguntes poden tenir dependències entre elles: per exemple, si un client ens diu que té telèfon mòbil, procedirem a preguntar-li el número d'aquest mòbil, però si ens diu que no, no li preguntarem el número. Així, el fet que el client tingui mòbil és un prerequisit per preguntar-li quin és el número de telèfon del seu mòbil.

Per això usem el sistema de regles que ens proporciona CLIPS i introduirem aquest prerequisit a la part esquerra de la regla.

```
1 (defrule preguntar-numero-mobil
2  (mobil si)
3  =>
4  (printout t "Quin és el teu número de mòbil?")
```

```
5   (bind ?numero-mobil (read))
6   (assert numero-mobil ?numero-mobil)
7  )
```

D'aquesta manera, la regla `preguntar-numero-mobil` només s'activarà si l'usuari ha contestat que sí a la pregunta "tens mòbil?".

4.3.1 Com puc ordenar aquest flux de preguntes?

Per altra banda, ens pot interessar fer que les preguntes que fem a l'usuari tinguin un cert ordre. Per exemple, ens pot interessar preguntar-li a l'usuari primer el seu nom, i posteriorment preguntar-li si té telèfon mòbil. En aquest cas la primera pregunta no és un prerequisit per preguntar la segona, però és de sentit comú que tinguin aquest ordre. Per traduir això a clips podem fer:

```
1  (defrule preguntar-mobil
2   (nom ?nom-usuari)
3   =>
4   (
5    if (yes-or-no-p "Tens telèfon mòbil?")
6    then (assert (mobil si))
7    else (assert (mobil no))
8   )
9  )
```

D'aquesta manera la pregunta sobre el mòbil només es dispararà quan l'usuari hagi contestat a la pregunta del seu nom.

4.3.2 I si vull saltar-me una pregunta?

En alguns casos ens podem trobar amb la situació de tenir una pregunta ja resolta implícitament en la resposta d'alguna pregunta prèvia. Per exemple, suposem que volem preguntar-li a una persona si vol prendre alguna beguda alcohòlica, però prèviament li hem preguntat la edat en aquesta persona. Podríem fer-ho de dues maneres.

1. Com fins ara, amb:

   ```
   1  (defrule preguntar-alcohol
   2   (edat major-edat)
   3   =>
   4   (
   5    if (yes-or-no-p "Vols prendre alguna beguda alcoholica?")
   6    then (assert (alcohol si))
   7    else (assert (alcohol no))
   8   )
   9  )
   ```

   Aquesta solució funcionaria, però imaginem que també hem preguntat a l'usuari quina és la seva religió. Si l'usuari ha contestat "religió musulmana", aleshores tampoc li haurem de preguntar si vol prendre alcohol. Haurem d'afegir a la part esquerra de la regla una sentencia com:

   ```
   (not (religio musulmana))
   ```

com a prerequisit, i haurem de fer el mateix amb cadascun dels fets o motius que impliquin que la persona no pot beure alcohol.

2. Una altra manera és "enganyar" al sistema i fer-li creure que l'usuari ja ha respost la pregunta sobre l'alcohol actuant en el moment de preguntar per la religió. Així:

```
(defrule preguntar-religio
 =>
 (bind ?religio
  (ask-question "Quina religió practiques?" catòlica musulmana budista cap))
 (if (eq ?religio "musulmana")
  then (assert (alcohol no)) (assert (porc no)))
```

I després:

```
(defrule preguntar-alcohol
 (not (alcohol ?si_o_no)
 =>
 ...
```

I ja posats també:

```
(defrule preguntar-porc
 (not (porc ?si_o_no))
 =>
 ...
```

D'aquesta manera aconseguim que el sistema no pregunti a l'usuari si vol prendre alcohol o si vol menjar porc, perquè en el moment de preguntar-li la religió hem tingut en compte aquests valors. A més, hem trobat una manera de implementar un flux de preguntes en el qual, alhora d'obtenir les respostes, no hem de modificar regles "futures" (dins del flux d'execució de preguntes).

4.3.3 Com inicialitzar el flux del programa?

Per inicialitzar el flux d'execució tenim varies possibilitats.

Algunes d'elles són:

- Definir un fet per defecte (deffact) el qual faci saltar una regla inicial per defecte (que podríem definir amb un salience alt).

  Per exemple,

```
(deffacts tipus-usuari
  (us desconegut)
)
```

  Que faria saltar la regla inicial:

```
(defrule inici
(declare (salience 10))
?us <- (us desconegut)
=>
(printout t "Benvingut!" crlf)
....
```

- Una regla inicial podria preguntar per si no existeix un fet que acabarà inicialitzant.

  Per exemple,

```
(defrule inici
 (not (flor ?tipus))
=>
(printout t "Benvingut" crlf)
...
(assert (flor rosa))
```

4.4 Uso de la función modify para ir guardando resultados preferidos

Para empezar hemos de tener una plantilla que, en el caso de un sistema para escoger platos, sea donde coloquemos los platos que se van ajustando a nuestra. Esta seria su estructura:

```
(deftemplate platos-apropiados
   (slot estado)
   (multislot lista-platos1)
   (multislot lista-platos2)
   (multislot lista-postres)
```

Entonces cuando vayamos obteniendo resultados para ir guardandalos en los diferentes multislots, tendremos que llamar a la instancia de dicha plantilla desde los activadores de las reglas, usando como guía el slot estado.

`?paprop <- (platos-apropiados (estado empezado))`

En este caso recuperariamos la instancia de `platos-apropiados` que ha sido inicializada en otra regla, pero mantendriamos la misma con los resultados ya guardados hasta el momento. Si en esta regla quisieramos insertar datos en otro multslot de dicha instancia (por ejemplo `lista-platos2`) tendriamos el siguiente código:

`(modify ?paprop (estado empezado)(lista-platos2 $?lista2))`

Donde `lista2` seria el resultado de hacer un `find-all-instances` por ejemplo.

4.5 Creación de la plantilla de recomendación

Una vez aplicadas todas las reglas necesarias y manipulado todo el conocimiento que hayamos tenido que usar en el problema, deberemos mostrar los resultados al usuario. Para ello, si partimos de que hemos tenido que construir un resultado a partir de ciertas preferencias o restricciones, lo más adecuado es usar una plantilla para poner el resultado y tenerlo estructurado. En este caso, lo que se debe hacer es partir de una plantilla vacia, la cual se irá rellenando conforme calculemos los resultados. La plantilla será un deftemplate con los atributos que va a tener el resultado final y deberá estar en un ámbito visible para todos los módulos, para que así la puedan ir modificando y completando.

Ejemplo de plantilla de recomendación para menús de restaurante:

```
(deftemplate recomendacion "Recomendacion resultante del sistema experto"
  (slot evento)
  (multislot menus)
  (slot final?)
```

Donde el slot evento guardaria una instancia de la clase Evento que contendria las propiedades de la comida (número invitados, temporada, ...), el multislot menus tendria las instancias de los menús finales para proponer y el slot final? seria una guia para, una vez obtenidos los resultados, activar la regla para mostrar los resultados por pantalla

5. Errores frecuentes

5.1 Quan obro un fitxer en CLIPS em dona un error!

L'editor de text de l'entorn gràfic de CLIPS no accepta fitxers més grans de 65KB. Aquesta quantitat de bytes pot ser àmpliament superada quan introduïm una gran quantitat d'instàncies a la base de coneixement mitjançant protégé i la exportem a format CLIPS.

Per saltar aquest problema només hi ha una solució: carregar el fitxer directament per línia de comandes, sense usar l'editor de text de CLIPS, amb la comanda `(load <NOM_FITXER>)`.

Per exemple:

```
(load fitxer.clp)
```

Per a continuació, fer:

```
(reset)
(run)
```

5.2 Codificació de caràcters a CLIPS

CLIPS dona força problemes amb caràcters estranys com ñ, accents, dièresis, etc.

En principi no accepta Unicode ni `ISO-8859-1`, així que recomanem no fer servir caràcters com els mencionats per tal de no tenir problemes. A més, protégé exporta amb codificació `ISO-8859-15`. La majoria d'editors de linux usen per defecte codificació `UTF-8`. Això és un problema, perquè alhora d'editar un document podeu perdre alguns caràcters si no configureu la codificació del vostre editor a `ISO-8859-15` i això provocarà que CLIPS no reconegui el fitxer com a vàlid. Un editor de linux que permet canviar la codificació de caràcters usada en un fitxer és `kwrite` (cal configurar-ho abans de començar a modificar el fitxer).

Si al carregar un fitxer tenim problemes i no ho sabem identificar, és útil comprovar si pot haver-hi algun caràcter que estigui creant aquests problemes.

5.3 ¿Por qué me dan error algunas de las restricciones que pongo en los slots en Protègè cuando las importo en CLIPS?

No todas las restricciones sobre los slots que se pueden definir en protègè están admitidas en la sintaxis de CLIPS. Tendréis problemas si ponéis un valor al número de posibles valores que puede tener un slot, en CLIPS sólo se puede indicar si un slot es obligatorio y si admite múltiples valores.

5.4 Tengo problemas con la heréncia de slots en las clases que he definido

A veces este problema aparece cuando se definen en protègè las superclases con *role* `abstract`. Si aparece se pueden definir todas las clases con *role* `concrete` y el problema debería arreglarse.

5.5 Què significa l'error OBJRTBLD5?

A l'apèndix G de la "CLIPS Basic Programming Guide" trobaràs el significat de tots els errors que dóna CLIPS.

5.6 Antes compilaba correctamente y ahora da warnings.

```
[CSTRCPSR1] WARNING: Redefining defrule: imprimir +j+j.
```

Si aparecen warnings como este a la hora de compilar en todas las funciones, reglas, etc, que se han definido en el código, es posible que no se haya hecho un `clear` antes de compilar nuevamente. Para ello usar la opción del menu *Execution* → *Clear Clips*, o bien, en la ventana de entrada de CLIPS escribir `(clear)`. Luego volvemos a cargar el fichero.

Este problema es debido a que los módulos no pueden ser ni redefinidos ni borrados una vez que se definen (con la excepción del módulo MAIN que puede ser redefinido una vez). La única forma de eliminar un módulo es con el comando clear.

5.7 No puedo editar mi fichero en clips.

Esto es debido a que el .clp ha llegado a su máxima extensión. Para solucionar el problema, debemos abrirlo con un editor de texto externo y proceder como siempre, es decir:

```
Execution -> Clear Clips
Execution -> Load (seguiremos cargando el .clp/.txt que
                   hemos modificado y guardado previamente en el editor de texto).
Execution -> Reset
Execution -> Run
```

5.8 Redefining

Este warning es debido en el mayor número de casos, a que existen dos o más reglas, funciones, ... que tienen el mismo nombre. Para solucionarlo, obviamente, cambiar el nombre de cada una de manera que no compartan nombres.

5.9 Problemas al consultar las instancias relacionadas con otras

Por alguna razón desconocida CLIPS solo busca instancias en el modulo actual y no en todos los importados. La solución más práctica es la que explica en 3.5.6. Otra solución es añadir a todas las instancias de la ontología el cualificador del modulo principal. Si se substituye en el fichero de las instancias la cadena "[" por "[MAIN::" estará todo solucionado.

Por ejemplo:

```
([pracIA_Instance_30001] of Bebida → ([MAIN::pracIA_Instance_30001] of Bebida
```

5.10 Unable to finde class X cuando definimos instancias

```
Defining definstances: instancias
[PRNTUTIL1] Unable to find class Plato.

ERROR:
(definstances MAIN::instancias
```

Si nos aparece algo parecido a esto cuando compilamos nuestro programa significa que no encuentra la clase a la que pertenece la instancia. Es posible que hayamos definido primero las instancias y luego las clases. Para corregirlo, seguir la estructura comentada en el fichero de CLIPS: ontología + instancias + código.

5.11 Expected the beginning of a construct (cuando definimos instancias).

Si nos aparece este error justo donde definimos las instancias, es posible que sea porque no hayamos seguido la correcta estructura para añadir instancias en nuestro código: `(definstances cualquiernombre INSTANCIAS )`.

5.12 Compila pero no compara bien dos elementos

Posiblemente se trata de que estemos realizando la comparación de dos cadenas de caracteres con el operador eq. Existe una funcion especifica para comparar dos strings: str-compare explicada en el apartado de funciones.

5.13 Expected the beginning of a constructor

Aquest error sol ser molt freqüent i es pot donar per diversos motius.

1. El primer i més freqüent és perquè hem posat un parèntesis tancat ")" de més. Surt quan, per exemple, tenim:

   ```
   (assert (processador AMD) ) ) ; <-- l'últim parèntesis sobra
   ```

   Haurem de repassar quin ha estat el codi que hem retocat que ens ha incorporat aquest error i trobar el parèntesis que sobra.

2. El segon motiu pel qual també es pot donar aquest error és perquè, per exemple, no hem encapsulat el codi de les instàncies generades automàticament mitjançant protégé dins la clau:

```
(definstances nominstancies
 <instàncies>
)
```

En definitiva, el que aquest error ens està indicant és que CLIPS espera la construcció d'una regla/funció/acció/etc. i en comptes d'això es troba amb qualsevol altra cosa: un parèntesis tancat, una instància, etc.

5.14 Missing function declaration for defrule/deffunction/...

Aquest error acostuma a donar-se quan ens hem descuidat de posar el parèntesis que tanca la definició d'una regla/funció/etc. i aquesta es "solapa" amb la següent regla/funció/etc. Per exemple:

```
(defrule pregunta-A
 (precondicio pre)
 =>
 (assert (accio realitzada))
     ; <--- aquí falta un parèntesis que tanqui la regla

(deffunction funcio-B
 ...
```

5.15 Check appropiate syntax for if/switch/loop-for-count/...

Aquest error ens apareixerà quan no hem respectat la sintaxi d'alguna de les estructures de control de flux com if, switch, loop-for-count, etc. Pot deure's a qualsevol motiu (falta de parèntesis, expressions incorrectes), però sempre dins de la estructura de control que ens indica CLIPS.

Per exemple:

```
(switch (?resposta) ; <---- sobren els parèntesis que engloben la ?resposta!!!
   (case 1  then (assert (fet primer)) )
   ...
)
```

5.16 Problemas con paréntesis

Se trata de uno de los errores más típicos a la hora de compilar y a su vez más difíciles de depurar. El problema se encuentra en que no hemos cerrado todos los paréntesis que se han abierto. Una forma rápida para comprobar si la forma de colocar los paréntesis es la correcta, seria sumar 1 cuando se abre un paréntesis y restar 1 cuando se cierra, de manera que si al final su cómputo es 0, es correcto. Por ejemplo:

```
    (  (  2  +  3 )  *  ( 4  /  ( 1 + 1 ) ) )
  0 1  2        1     2      3        2 1 0
```

Es muy recomendable y facilita mucho la tarea de encontrar bugs, ponerlos tabulados con comentario en el cierre

```
(while
...
) ;endwhile
```

6. Referencias

6.1 On puc trobar informació sobre el llenguatge CLIPS?

Llegeix-te la "CLIPS User Guide". És un document que explica amb un llenguatge senzill tot el que es pot fer amb CLIPS, començant amb coses fàcils i es va complicant mica en mica. Veuràs que és una mica llarga, però és prou entretinguda de llegir. La pots trobar a l'adreça següent:

https://clipsrules.net/documentation/v640/ug640.pdf.

No obstant, a la "CLIPS User Guide" només hi ha els conceptes més importants. Si no trobes alguna cosa, consulta la "CLIPS Basic Programming Guide", que trobaràs aquí:

https://clipsrules.net/documentation/v640/bpg640.pdf.