Dpto. Sistemas Informáticos y Computación
Escuela Técnica Superior de Ingeniería Informática
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

SISTEMAS INTELIGENTES

Seminario de CLIPS (V6.3)

(C Language Integrated Production System)

CLIPS V6.3

(C Language Integrated Production System)

1. Introducción

CLIPS es una herramienta de Sistemas Expertos desarrollada orignalmente por Software Technology Branch de la NASA/Lyndon B. Johnson Space Center. CLIPS está diseñado para la construcción de Sistemas Basados en Reglas (SBR) y facilitar el desarrollo de software que requiere modelizar conocimiento de expertos en un determinado problema.

Hay tres formas de representar conocimiento en CLIPS:

- **reglas**: especialmente destinadas para representar conocimiento heurístico basado en la experiencia
- **funciones**: para representar conocimiento procedural
- **programación orientada a objetos (POO)**: para representar principalmente conocimiento procedural. CLIPS soporta las 6 características generlamente aceptadas de la POO: clases, paso de mensajes, abstración, encapsulamiento, herencia y polimorfismo.

CLIPS soporta por tanto los 3 paradigmas de programación:

- **programación basada en reglas**: reglas+ hechos (facts)
- **programación procedural:** funciones
- **programación orientada a objetos**: objetos+paso de mensajes

De los tres paradigmas, nosotros solo utilizaremos la programación basada en reglas y, puntualmente, la programación procedural.

Características de CLIPS:

- CLIPS es una herramienta escrita en C
- Plena integración con otros lenguajes como C y ADA
- Desde un lenguaje procedural se puede llamar a un proceso CLIPS, éste realiza su función y luego le devuelve el control al programa.
- Se puede incorporar código procedural como funciones externas en CLIPS
- Las reglas pueden hacer "*pattern-matching*" sobre objetos y hechos formando así un sistema integrado.
- CLIPS tiene una sintaxis estilo LISP que utiliza paréntesis como delimitadores.

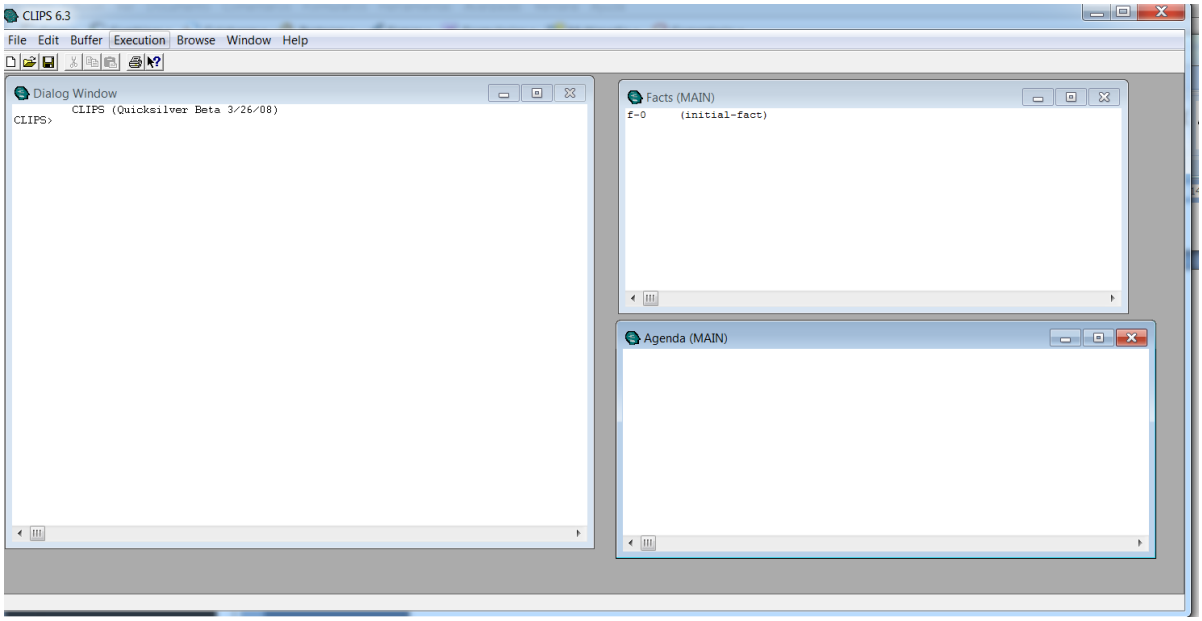
CLIPS es un Sistema de Producción que se compone de los siguientes módulos:

- Memoria Global o **Base de Hechos** (hechos + instancias de objetos)
- **Base de Reglas** ó Base de Conocimientos (reglas)
- **Motor de Inferencia** (control)

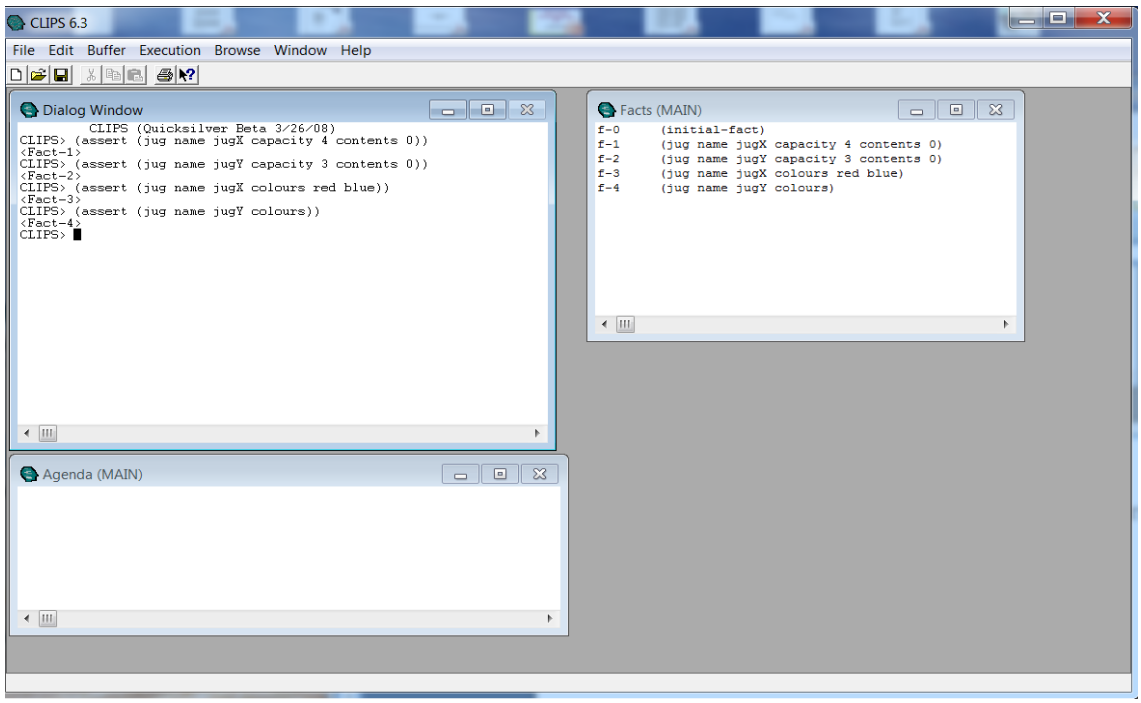
2. Interfaz de CLIPS

La interfaz de CLIPS se compone de tres ventanas principales:

- **Dialog Window**. Intérprete de CLIPS ó ventana del símbolo del sistema donde se puede teclear comandos ejecutables que CLIPS evaluará.
- **Facts**. Base de Hechos que contiene los hechos del problema
- **Agenda.** Agenda o Conjunto conflicto donde se almacenarán las instancias de reglas o activaciones.



Ejemplos de utilización del comando `assert`.

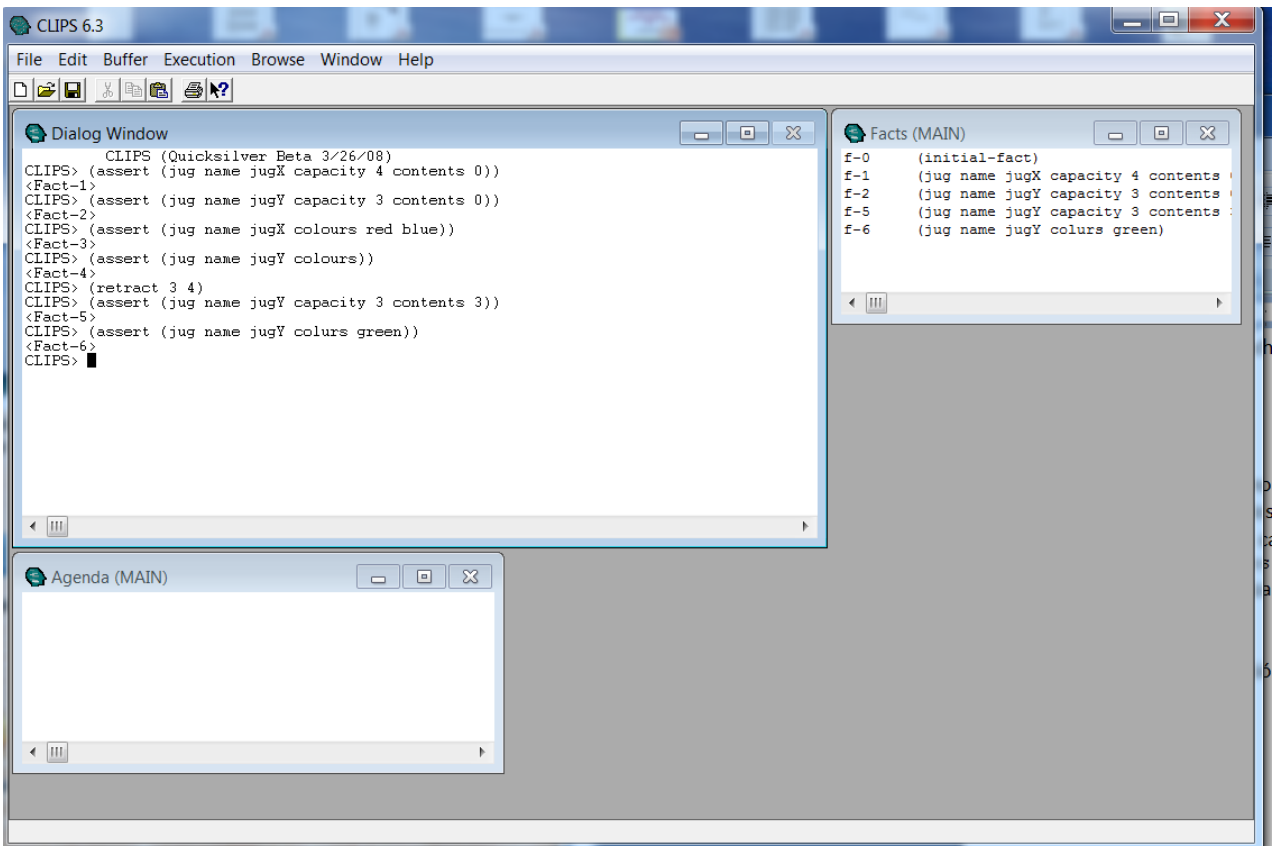


- El comando **`assert`** se utiliza para insertar un hecho en la Base de Hechos (BH). La sintáxis es `(assert <fact>+)` .
- Los hechos representan un conjunto de variables, propiedades y valores que éstas pueden tomar.
- Un hecho se compone de número ilimitado de campos separados por blancos y encerrados entre paréntesis balanceados. Los campos pueden tener o no un nombre asignado.
- Todos los hechos tienen un identificador ó índice de la forma **`f-n`** (ver ventana **Facts**). donde '**n**' es el índice del hecho (`fact-index`).
- CLIPS define por defecto un hecho inicial `(initial-fact)` cuando se carga la aplicación.
- Todos los hechos se insertan en la lista ***Facts Window*** (BASE DE HECHOS).

Hechos ordenados:

- Los campos no tienen nombre asignado
- El orden de los campos es significativo
- Tipos de los campos: `float, integer, symbol, string, external-address, fact-address, instance-name, instance-address`
- Se suele utilizar el primer campo de un hecho para describir una relación entre campos.

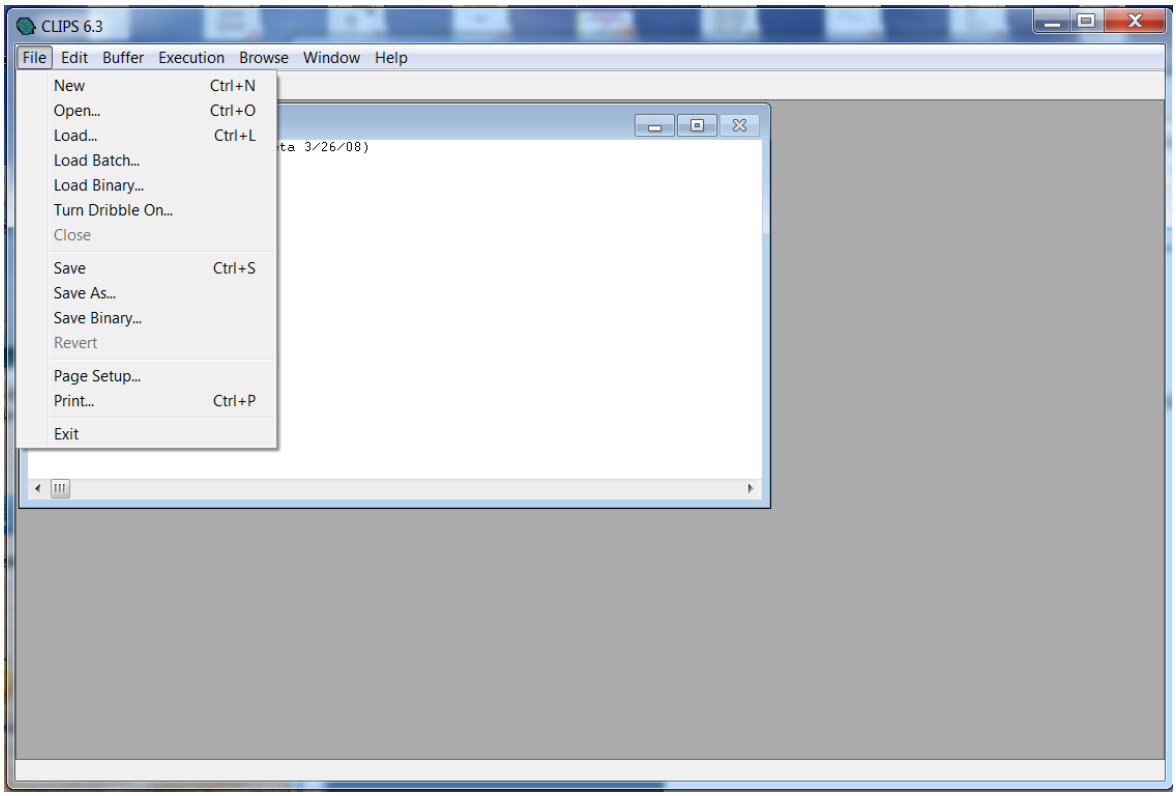
Ejemplos de utilización del comando `retract`.



- El comando `retract` se utiliza para eliminar un hecho de la BH. La sintáxis de este comando es `(retract <fact-index>+)`.
- Cuando se elimina un hecho, el resto conserva su índice o identificador original
- Cualquier nuevo hecho que se inserte en la BH se le asigna el índice siguiente al del último hecho insertado

2.1 Entorno CLIPS

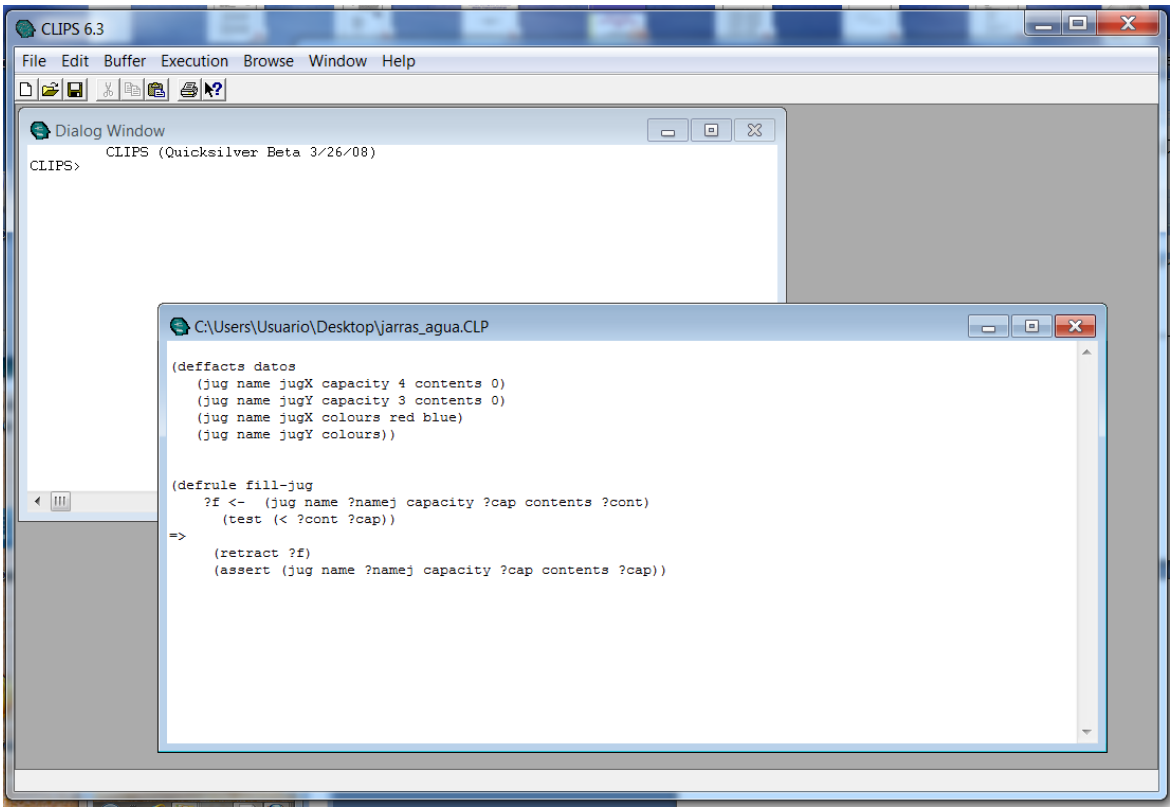
Escribe tu SBR en un fichero de texto. Puedes utilizar cualquier editor de texto o bien el editor de CLIPS (opción **New** del menú **File**).



A continuación se muestra un ejemplo de un SBR que solo contiene un conjunto de hechos iniciales, definidos mediante el constructor `deffacts` y una regla definida mediante el constructor `defrule`.

Mediante el constructor `deffacts` se definen todos los hecho iniciales del problema. El constructor `deffacts` simplemente almacena en memoria la estructura definida para introducir los hechos en el momento de ejecutar el SBR. La sintáxis es

```
(deffacts <statement-name>
        (<fact-1>)
        (<fact-2>)
          ……..   )
```



Las tres fases para poner en marcha un SBR son:

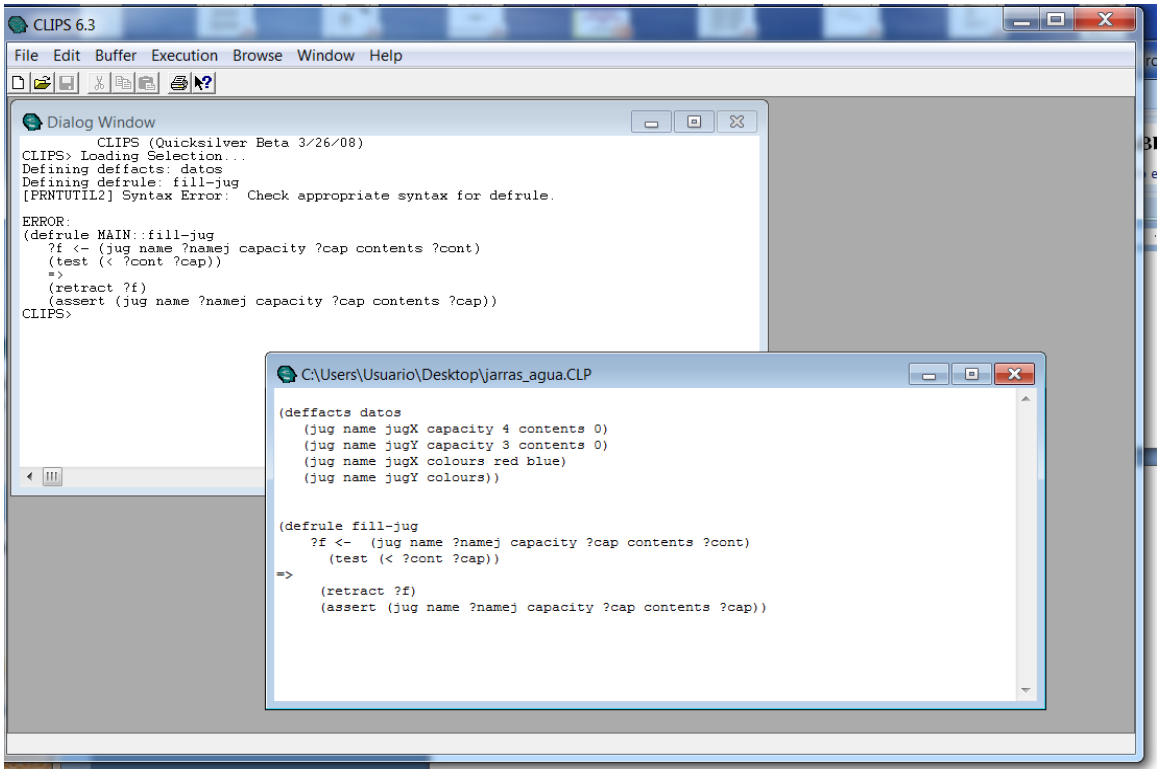
- **Cargar el fichero**. Esta operación realiza un parsing del fichero para comprobar que no hay errores sintácticos.
- **Reset CLIPS**. Vacía el contenido actual de la BH y la agenda; carga los hechos definidos en la estructura deffacts, las reglas definidas mediante defule y realiza la primera fase de *matching*.
- **Ejecutar el SBR**. Esta operación pone en marcha el Motor de Inferencia y ejecuta el SBR, aplicando sucesivamente el ciclo reconocimiento-acción del algoritmo RETE.

Cargar fichero

Después de guardar el fichero, la siguiente operación es cargar el fichero. Para ello se puede hacer de dos formas:

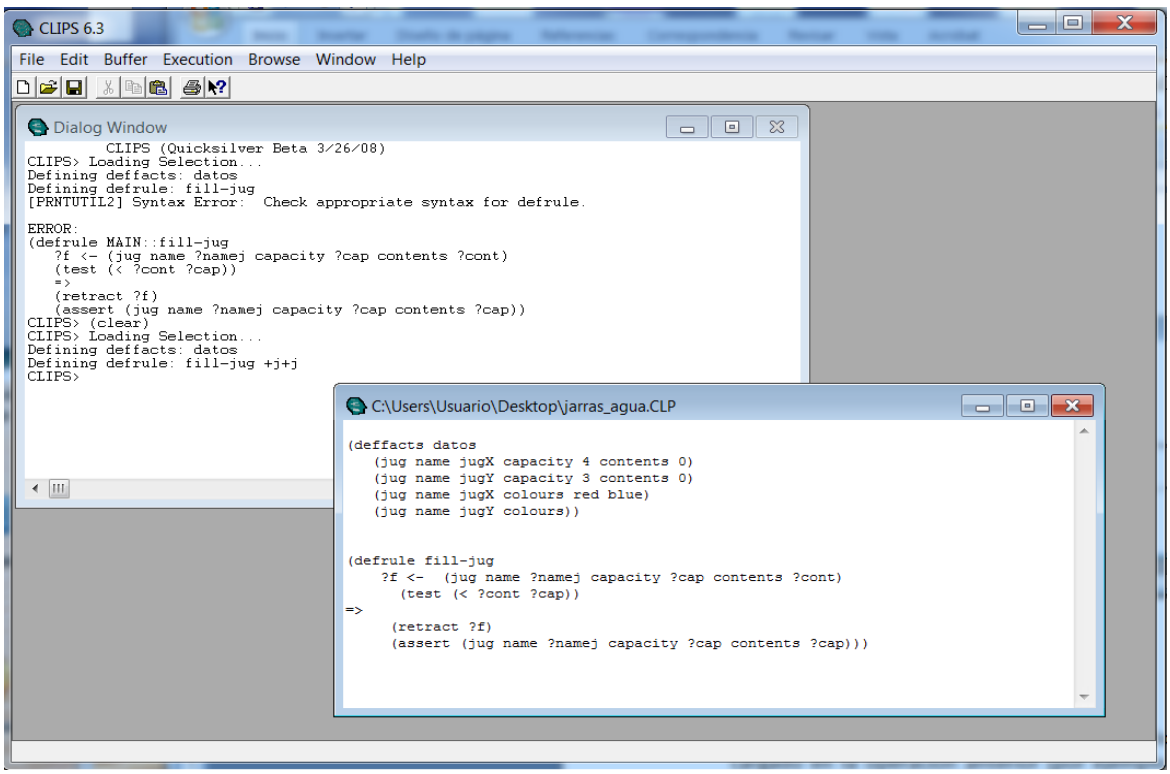
- Utilizando la opción **Load …** del menú **File**
- Utilizando la opción **Load Buffer** del menú **Buffer**

Esta operación comprueba que todas las estructuras CLIPS del fichero están correctamente escritas. En el ejemplo que se muestra a continuación, se puede observar que hay un error en la definición de la regla; concretamente, el error se sitúa al final de la regla, donde falta un paréntesis de cierre.



En este caso, lo que debe hacerse es:

a. Corregir el error del fichero
b. Teclear el comando `(clear)` para borrar todos las estructuras que pudieran haberse cargado en la operación anterior (por ejemplo, la estructura `deffacts`). El comando `(clear)` también puede activarse desde la opción **Clear CLIPS** del menú **Execution**
c. Volver a cargar el fichero



Reset CLIPS

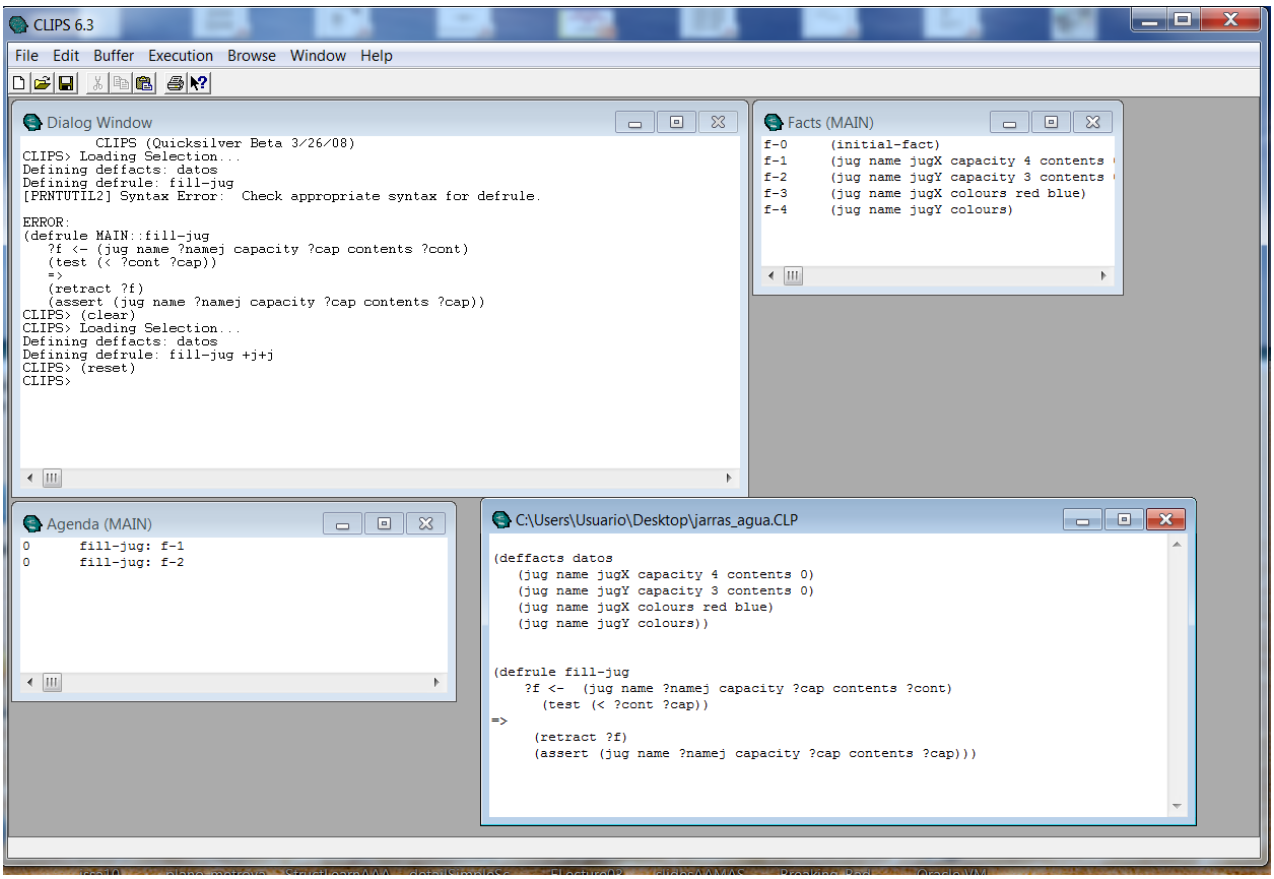
Una vez que el fichero está libre de errores, la siguiente operación es cargar las estructuras de datos en memoria. Concretamente, el comando `(reset)` realiza las siguientes operaciones:

1. Borra los hechos existentes en la BH de la ejecución anterior.
2. Inserta el hecho inicial `(initial-fact)`.
3. Inserta los hechos definidos en los constructores `deffacts` en la BH
4. Realiza la primera fase de *matching* construyendo las posibles activaciones de reglas que puedan existir.

El comando `(reset)` puede activarse:

a. Directamente en la ventana de comandos (Dialog Window)
b. Mediante la opción **Reset** del menú **Execution**

Como puede observarse en la siguiente pantalla, la operación `(reset)` ha introducido los hechos en la BH y dos activaciones en la Agenda (la regla `fill-jug` para la jarra de nombre `jugX`, y la misma regla `fill-jug` para la jarra de nombre `jugY`).



Ejecutar CLIPS

El tercer y último paso es ejecutar el SBR. Esta operación puede realizarse de dos formas:

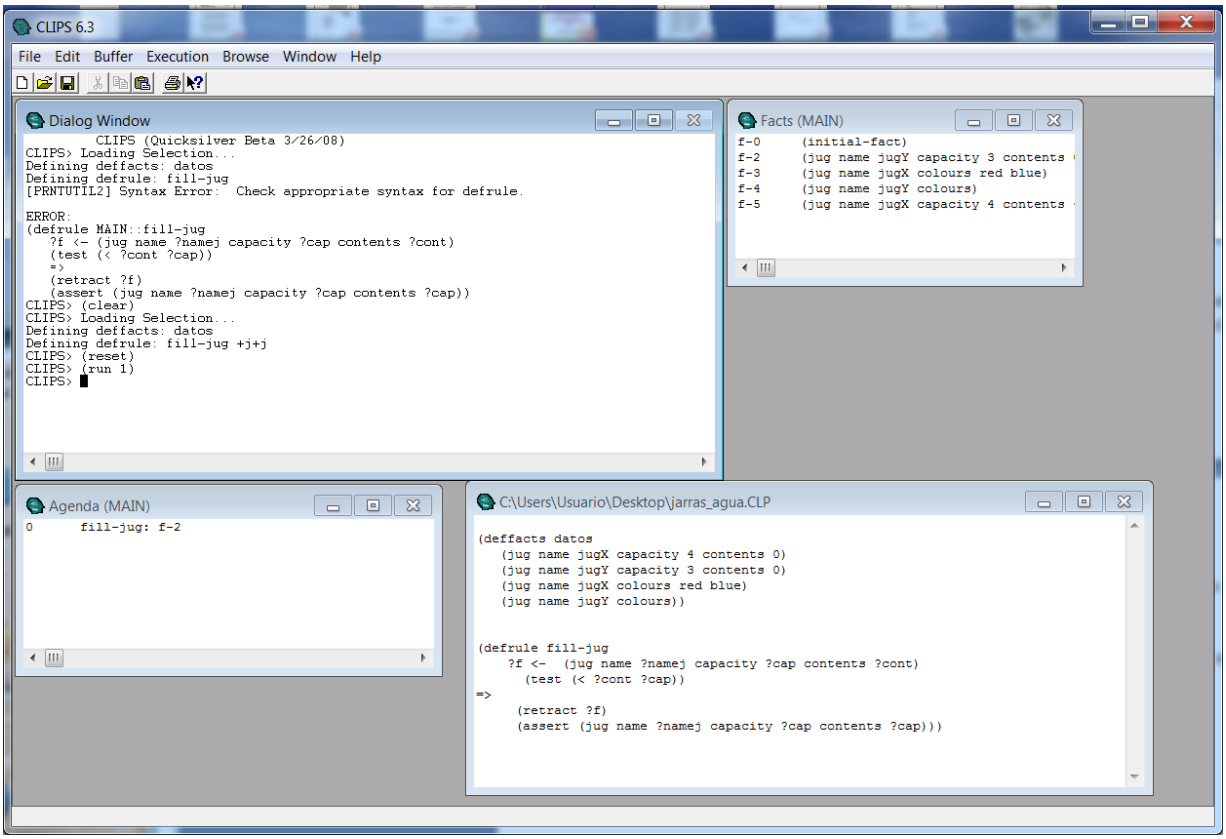
a. Tecleando el comando `(run)` o seleccionado la opción **Run** del menú **Execution**. El comando Run ejecuta sucesivamente el ciclo match-selección-ejecución hasta que el sistema finaliza (ó bien una regla para el proceso inferencial, o bien la agenda se queda vacía)

b. Mediante la opción **Step** del menú **Execution.** Esta opción permite ir ejecutando el SBR paso a paso, viendo cada ciclo del proceso inferencial.

Seleccionamos la opción **Step** en nuestro problema. Cada operación Step realiza las siguientes fases:

1. selecciona la primera activación de la Agenda
2. ejecuta la parte derecha de la instancia seleccionada (RHS de la regla seleccionada)
3. realiza de nuevo el proceso de *matching*.

En nuestro pequeño ejemplo, la ejecución de la primera instancia de la agenda borra el hecho **`f-1`** y genera el hecho **`f-5`** (que representa que la jarra `jugX` está ahora llena) pero el hecho **f-5** no provoca ningún nuevo *matching* porque no activa la única regla que tenemos definida en el SBR.



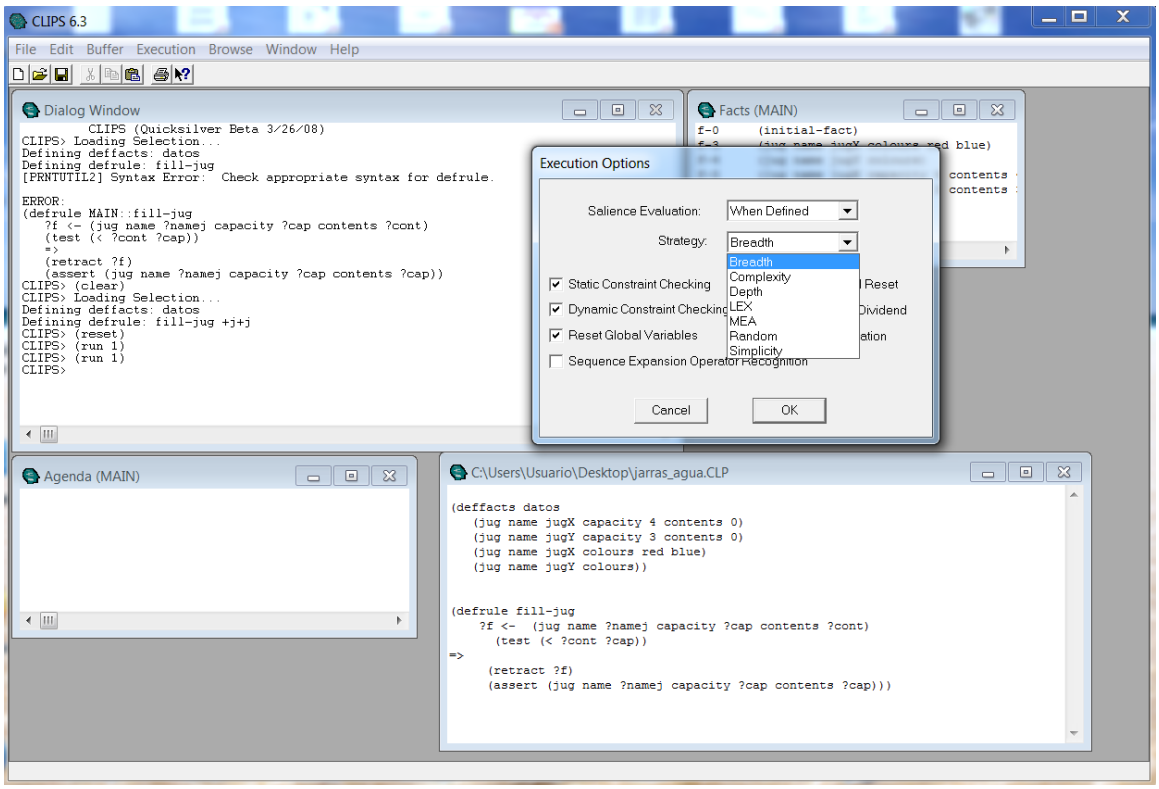
A continuación volveríamos a ejecutar **Step** del menú **Execution**, se seleccionaría la única instancia disponible en la Agenda, se ejecutaría su RHS y generaría un nuevo hecho **`f-6`** que nono instancia la regla `fill-jug`, por tanto, la fase de matchig no genera nuevas activaciones. El proceso inferencial se pararía entonces en este punto.

2.2 Estrategias de Control

Para una definición detallada de las ***estrategias de resolución de conflictos*** en CLIPS (refracción, no duplicidad de hechos, etc.), ver transparencias del Bloque 1, Tema 2.

Básicamente, en nuestros ejemplos trabajaremos con las siguientes estrategias de control: Anchura (**Breadth**), Profundidad (**Depth**) y, eventualmente, una estrategia aleatoria (**Random**).

Para seleccionar una estrategia de control en CLIPS, hay que ir a la opción **Options** del menú **Exeution**. En esta nueva pantalla, pinchar en el menú desplegable de **Strategy** y ahí encontraremos las opciones de estrategias de control disponibles en CLIPS.



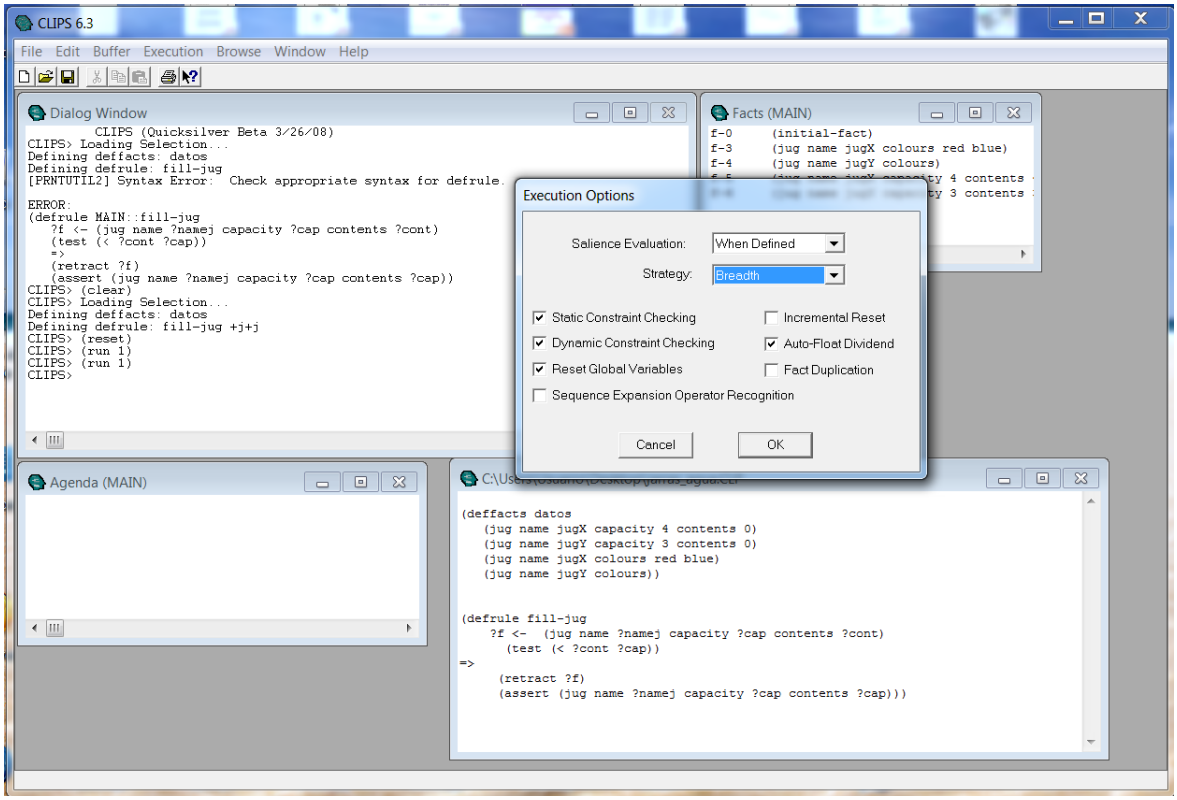
Asimismo, puede observarse que en la pantalla de **Execution Options** aparece la etiqueta **Fact Duplication** sin marcar (opción por defecto de CLIPS).

Recordatorios:

1. CLIPS utiliza refraccion: no permite activar una regla más de una vez con los mismos hechos (hechos con los mismos números de índice).

2. CLIPS por defecto no permite Fact Duplication

Para más detalles, ver transparencias del Bloque 1, Tema 2.



3. Motor de inferencia de CLIPS

En este apartado, se presentan dos ejemplos de SBR para analizar el comportamiento del ciclo reconocimiento-acción (ó ciclo *matching*-selección-ejecución) basado en el algoritmo Rete de *matching*.

3.1 Ordenación de una lista de números

Sea una BH que contiene un hecho que representa una lista de números naturales no ordenados. Escribir una única (si es posible) regla de producción que obtenga, como BH final, la lista inicial con sus elementos en orden creciente:

Ejemplo: (lista 4 5 3 46 12 10) obtiene: (lista 3 4 5 10 12 46)

El fichero CLIPS sería el siguiente (se puede descargar este código del fichero `OrdenarNumeros.clp`).

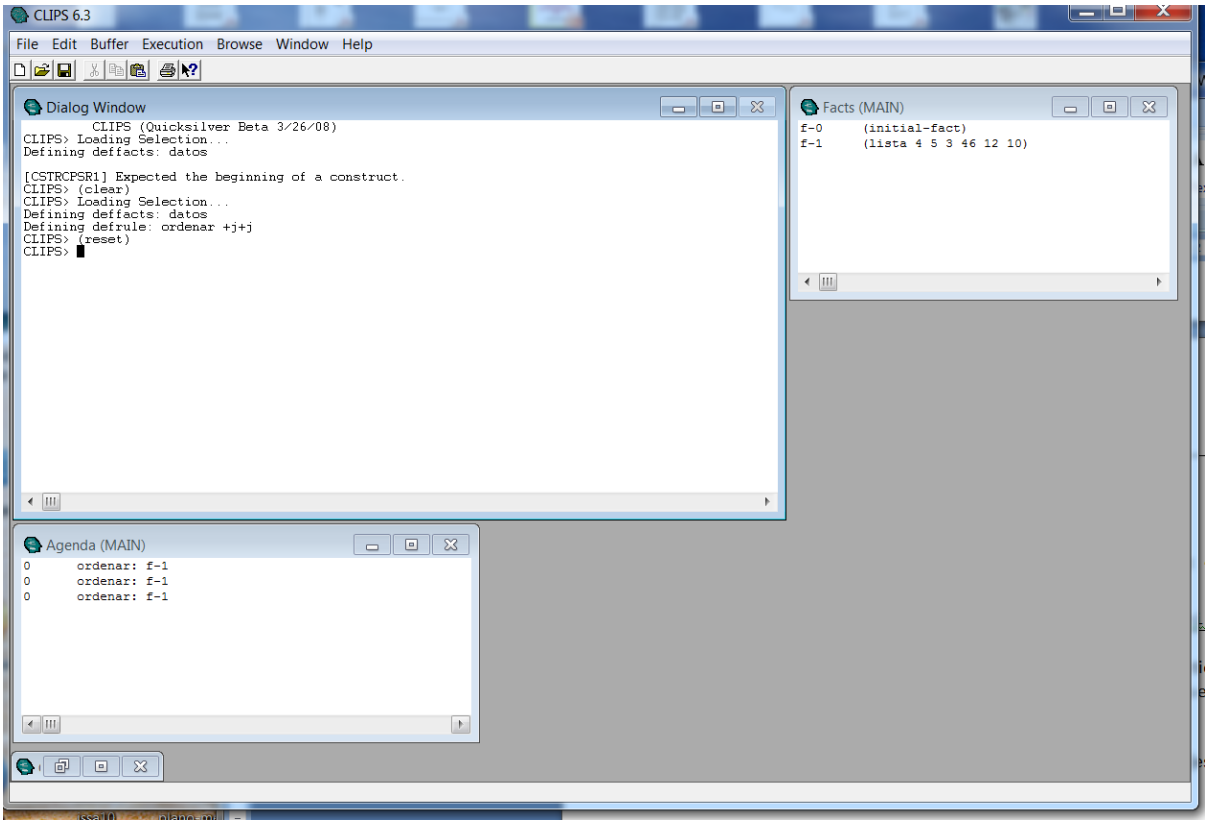
```
(deffacts datos
     (lista 4 5 3 46 12 10))

(defrule ordenar
   ?f1 <- (lista $?x ?y ?z $?w)
          (test (< ?z ?y))    ;; comprobamos si ?z es menor que ?y
  =>
      (retract ?f1)
      (assert (lista $?x ?z ?y $?w)))  ;; intercambiamos elementos
```

En este ejemplo, cuando el Motor de Inferencia pare será porque no tiene más instancias de la regla `ordenar` en la agenda, lo que significa que ya estarán todos los elementos de la lista original ordenados.

Seleccionamos **Breadth** (Anchura) como estrategia de resolución de conflictos (este problema se corresponde con el ejemplo 2 del tema 7).

Tras cargar el fichero, y teclear `(reset)`, obtenemos el siguiente resultado:



Se puede observar que en la Agenda hay tres activaciones de la regla `ordenar`. Todas ellas instancian con el único hecho disponible en la BH:

1. La primera instancia corresponde con el intercambios de los valores 12 10.

2. La segunda instancia corresponde con el intercambios de los valores 46 12.
3. La tercera instancia corresponde con el intercambio de los valores 5 3.

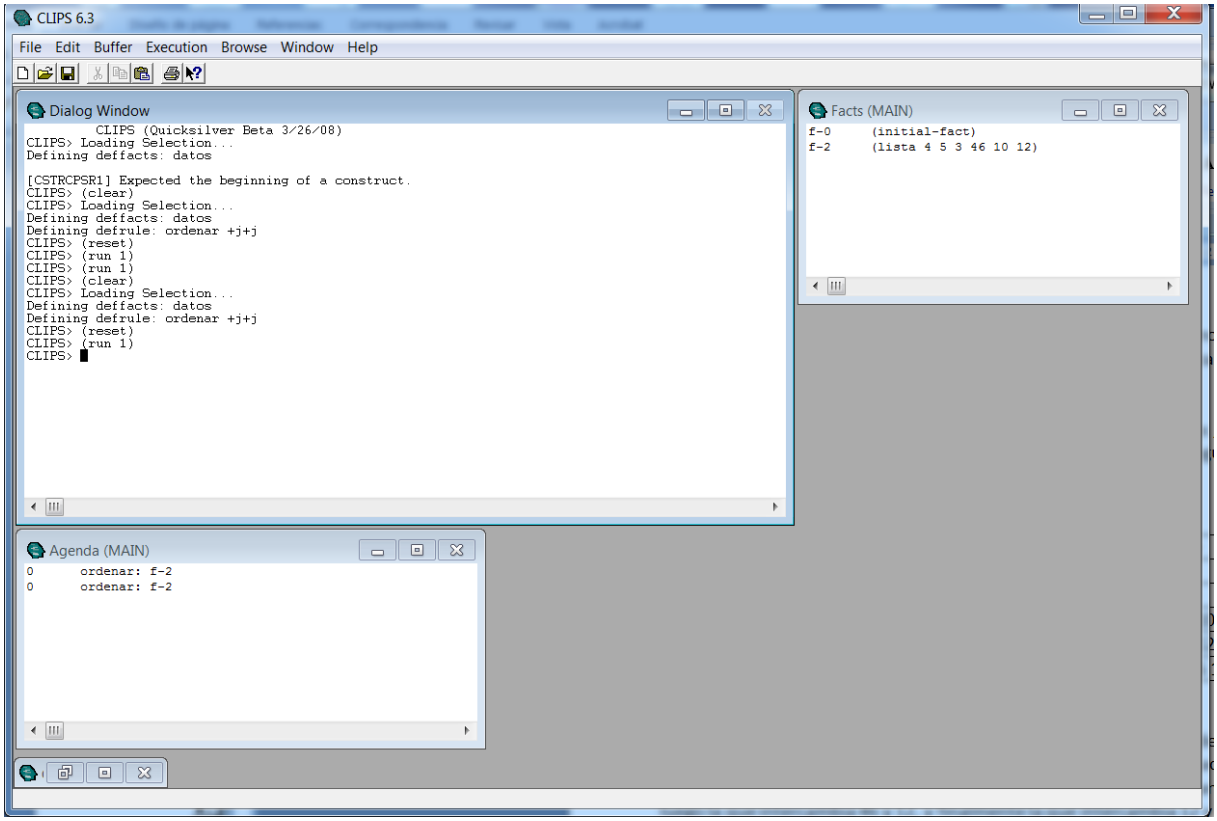
El orden en el que la estrategia de ANCHURA inserta las activaciones encontradas en la fase de matching de un ciclo tiene que ver con el orden en el que CLIPS realiza el proceso de *pattern-matching*. CLIPS comienza por la última regla del SBR, último patrón de la LHS de la regla, ó última variable multi-valuada de un patrón de la regla.

De este modo, tenemos que CLIPS intenta primero ligar cero elementos a la variable multi-valuada `$?w` (ó, equivalentemente, todos los posibles elementos a la variable `$?x`); a continuación liga un solo elemento a la variable `$?w`, después dos elementos, y así sucesivamente. Concretamente, el orden en el que CLIPS hace el *pattern-matching* es:

#Match	$?x	?y	?z	$?w	Test
1	(4 5 3 46)	12	10	()	**TRUE**
2	(4 5 3)	46	12	(10)	**TRUE**
3	(4 5)	3	46	(12 10)	FALSE
4	(4)	5	3	(46 12 10)	**TRUE**
5	()	4	5	(3 46 12 10)	FALSE

La estrategia de ANCHURA introduce las activaciones en el mismo orden en el que CLIPS realiza el *pattern-matching* porque añade cada nueva activación al final. En cambio, la estrategia de PROFUNDIDAD introduciría las activaciones en el orden inverso al mostrado en la tabla porque añade cada nueva activación al principio; es decir, primero introduciría la instancia que intercambia 5 y 3, luego la que intercambia 46 y 12, y finalmente la que intercambia 12 y 10.

Siguiendo con la estrategia en ANCHURA … ejecutamos el primer ciclo `(step)` y el resultado es:



- Se elimina el hecho `f-1`; como consecuencia de esto, se eliminan también de la Agenda las otras dos instancia que dependían de `f-1`
- Se genera un nuevo hecho `f-2` donde los elementos 12 y 10 ya están intercambiados
- Automáticamente, se lleva a cabo la siguiente fase de *matching* y aparecen dos activaciones con el hecho `f-2` (la primera corresponde a los elementos 46 y 10, y la segunda a los elementos 5 y 3).

Continúa la ejecución del SBR paso a paso hasta que la Agenda se queda vacía y comprueba el estado final de la BH.

3.2 Secuencias de ADN

Dadas dos secuencias de ADN, escribe un SBR (una única regla, si es posible) que cuente el número de mutaciones (se puede utilizar una variable global para almacenar el número de aciertos o bien un hecho para registrar este valor). Por ejemplo, para las dos secuencias de ADN

(A A C C T C G A A A) y **(A G G C T A G A A A)** hay tres mutaciones.

El fichero CLIPS sería el siguiente (se puede descargar este código del fichero `MutacionesADN.clp`).

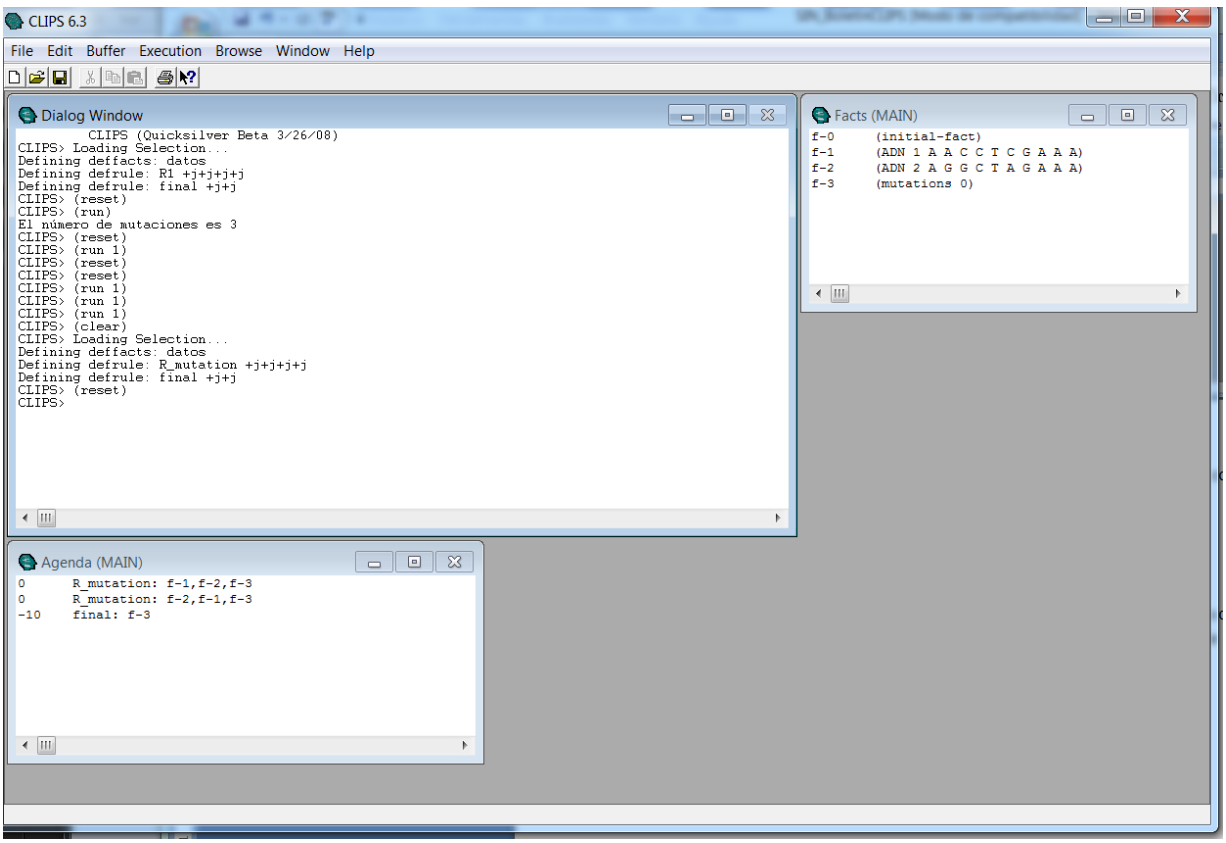
En este caso, hemos definido una regla `final` que muestra por pantalla el resultado. Le damos a la regla final una prioridad menor para que se lance únicamente cuando no haya activaciones de la regla `R_mutation`).

```
(deffacts datos
     (ADN 1 A A C C T C G A A A)
     (ADN 2 A G G C T A G A A A)
     (mutations 0))

(defrule R_mutation
  ?f1 <-   (ADN ?n1 $?x ?i1 $?y1)
  ?f2 <-   (ADN ?n2 $?x ?i2 $?y2)
  ?f3 <-   (mutations ?m)
     (test (and (neq ?n1 ?n2)(neq ?i1 ?i2)))
  =>
     (retract ?f1 ?f2 ?f3)
     (assert (ADN ?n1 $?y1))
     (assert (ADN ?n2 $?y2))
     (assert (mutations (+ ?m 1))))

(defrule final
  (declare (salience -10))
   (mutations ?m)
 =>
   (printout t "El número de mutaciones es " ?m crlf))
```

Seleccionamos **Depth** (Profundidad) como estrategia de resolución de conflictos. Tras cargar el fichero, y teclear `(reset)`, obtenemos el siguiente resultado:



Las dos instancias que aparecen en la Agenda se corresponden realmente con la misma mutación; esto se debe a que:

- `f-1,f-2,f-3` hacen *matching* con la regla R_mutation, y
- `f-2,f-1,f-3` también hacen *matching* con la regla R_mutation

No obstante, cuando una activación se lance y ejecute su RHS, borrará los hechos que han instanciado los patrones eliminando automáticamente la otra instancia dependiente de los mismos hechos.

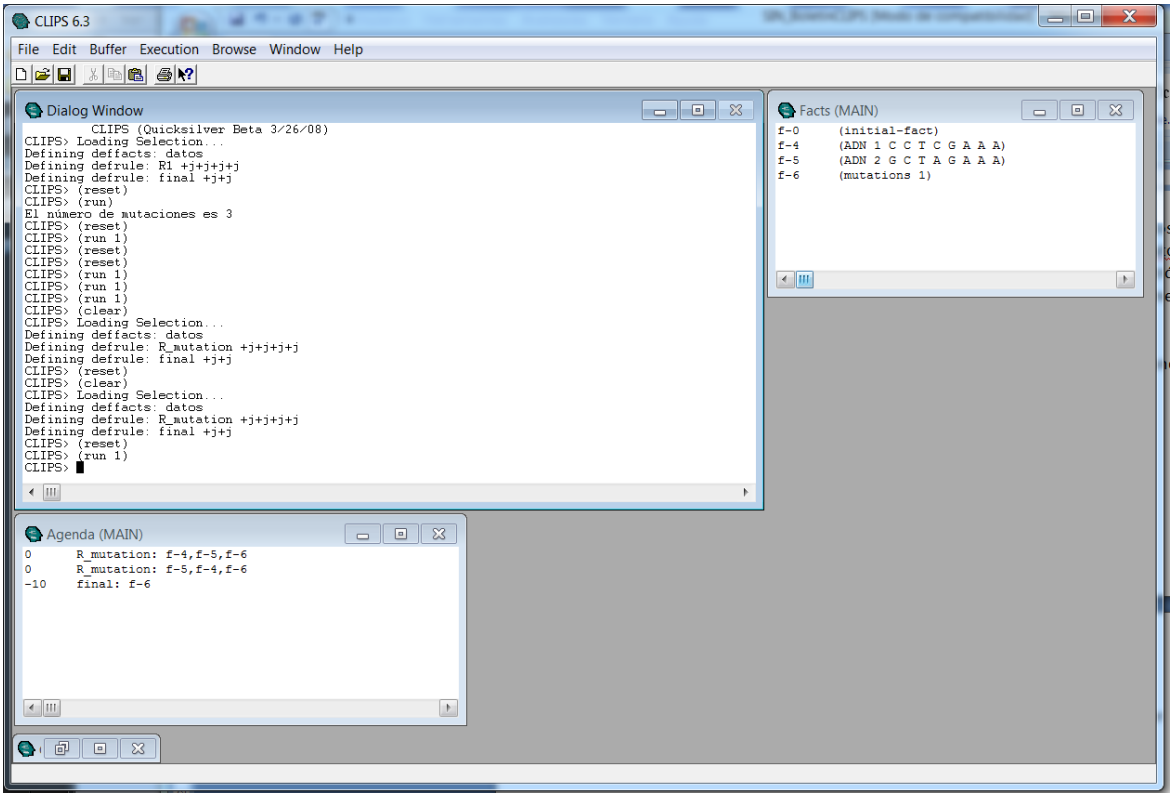
Como se ha explicado anteriormente, el orden en el que se introducen las dos activaciones en la Agenda depende del orden en el que CLIPS realiza el proceso de *pattern-matching* y de la estrategia de resolución seleccionada:

1. CLIPS comienza por el tercer patrón, instanciándolo con `f-3`; luego instancia el segundo patrón con el primer hecho que encuentra `f-1;` consecuentemente, el primer patrón se instancia con el hecho `f-2` para que el test sea TRUE.
2. A continuación intenta buscar otra combinación de hechos que instancie la LHS de la regla: tercer patrón con `f-3`, segundo patrón con el siguiente hecho que encuentra, `f-2`, y, consecuentemente, el primer patrón con el hecho `f-1` para que el test sea TRUE.

Las dos activaciones que aparecen en la Agenda, y que en realidad corresponden a la misma mutación, son:

#Match	**?n1**	**?n2**	**$?x**	**?i1**	**?i2**
1	1	2	(A)	A	G
2	2	1	(A)	G	A

Siguiendo con la estrategia en PROFUNDIDAD … ejecutamos el primer ciclo `(step)` y el resultado es:



- Se eliminan los hechos `f-1, f-2` y `f-3`; como consecuencia de esto, se elimina también de la Agenda la otra activación
- Se genera tres nuevos hechos `f-4, f-5` y `f-6`
- Automáticamente, se lleva a cabo la siguiente fase de *matching* y aparecen dos activaciones correspondientes a la mutación de los primeros elementos de cada secuencia de ADN (C de la secuencia 1 y G de la secuencia 2).

Continúa la ejecución del SBR paso a paso hasta que se dispare la regla final y muestre por pantalla el número total de mutaciones. Puedes probar este mismo SBR con otras dos secuencias de ADN.

ANEXO: Aspectos adicionales del lenguaje CLIPS

Los principales elementos del lenguaje CLIPS se han visto y utilizado en las transparencias correspondientes a los temas 6 y 7 del bloque 3 Representación del Conocimiento. En este apartado presentamos algunas elementos adicionales de CLIPS que pueden ser de utilidad para el diseño y desarrollo de un SBR.

Comando `printout`

Este comando permite obener una salida en el dispositivo representado por `<logical-name>`.

```
(printout <logical-name> <expression>*)
```

El dispositivo de salida puede ser un fichero ó bien la pantalla. El símbolo **t** es el que se utiliza para indentificar la salida estándar. Ejemplos:

```
CLIPS> (printout t "Hello there!" crlf)
Hello There!
CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS>
```

`crlf` signfica carriage return line feed (retorno de carro)

Otro ejemplo de utilización del comando `printout` en la RHS de una regla para mostrar resultados por pantalla.

```
(defrule find-data-1
      (data ?x $?y ?z)
=>
      (printout t "?x = " ?x crlf
                  "?y = " ?y crlf
                  "?z = " ?z crlf
                  "------" crlf))
```

Comando `bind`

Este es el comando que se utiliza en CLIPS para ligar un valor a una variable, monovaluada ó multivaluada: `(bind <variable> <expression>*)`

Ejemplos:

```
(defrule roll-the-dice
      (roll-the-dice)
```

```
=>
      (bind ?roll1 (random 1 6))
      (bind ?roll2 (random 1 6))
      (printout t "Your roll is: " ?roll1 " " ?roll2 crlf))


(defrule drop-all-water-from-Y-to-X
    ?f1 <-  (jug name ?namex capacity ?capx contents ?contx)
    ?f2 <-  (jug name ?namey capacity ?capy contents ?conty)
      (test  (and (> ?conty 0) (< ?contx ?capx) (neq ?namex ?namey)))
        (test (< (+ ?contx ?conty) ?capx))
=>
     (retract ?f1 ?f2)
     (bind ?newcont (+ ?contx ?conty))
     (assert (jug name ?namex capacity ?capx contents ?newcont))
     (assert (jug name ?namey capacity ?capy contents 0)))
```

Variables globales

Las variables globales deben definirse al principio del programa mediante el comando **`defglobal`**. El nombre de una variable global siempre debe ir encerrado entre dos astericos (*).

Ejemplo de declaración de variables globales:

```
      (defglobal ?*lista_a* = (create$))
      (defglobal ?*valor* = 0)
```

Se puede ligar un valor a una variable al mismo tiempo que se define dentro del comando defglobal.

Otro ejemplo:

```
CLIPS> (defglobal ?*x* = 3)
CLIPS> ?*x*
3
CLIPS> red
red
CLIPS>(bind ?a 5)
5
CLIPS> (+ ?a 3)
8
CLIPS> (reset)
CLIPS> ?a
[EVALUATN1] Variable a is unbound
FALSE
CLIPS>
```

Comando `deffunction`

Al igual que otros lenguajes, CLIPS permite definir funciones propias mediante la primitiva **`deffunction`**. Las funciones definidas con `deffunction` tienen un ámbito global y son

llamadas del mismo modo que cualquier otra función. Además estas funciones se pueden utilizar como argumentos de otras funciones.

```
(deffunction <function-name> [optional-comment]
      (?arg1 ?arg2 … ?argM [$?argN])
      (<action1>
      <action2>
      ……
      <action (k-1)>
      <action k>)
```

1) $?arg_i$: parámetros
2) la función sólo devuelve el valor de la última acción <action k>. Esta acción puede ser una función, una variable o una constante.

```
(deffunction hipotenusa
      (?a ?b)
      (sqrt (+ (* ?a ?a)(* ?b ?b))))
```

```
(defrule calcula-hipotenusa
   (dimensiones ?base ?altura)
 =>
   (printout t "Hipotenusa =" (hipotenusa ?base ?altura) crlf))
```

Programación procedural

Se puede incluir código procedural en la parte derecha de una regla. Estas estructuras son:

- while
- if then else
- break
- return

Ejemplo:

```
(deffunction inicio ()
   (reset)
   (printout t "Profundidad Maxima:= " )
   (bind ?*prof* (read))
   (printout t "Tipo de Busqueda " crlf "
               1.- Anchura" crlf "    2.- Profundidad" crlf )
   (bind ?a (read))
   (if (= ?a 1)
       then    (set-strategy breadth)
       else   (set-strategy depth))
  (printout t " Ejecuta run para poner en marcha el programa " crlf))
```

Funciones de predicado

Funciones de predicado que se pueden usar en los tests condicionales de las reglas.

(**evenp** <arg>): TRUE si el número es par
(**floatp** <arg>): TRUE si es un número en coma flotante
(**integerp** <arg>): TRUE si es un número entero
(**lexemep** <arg>): TRUE si el argumento es un símbolo o un string
(**numberp** <arg>): TRUE si el argumento es un número
(**oddp** <arg>): TRUE si el número es impar
(**pointerp** <arg>): TRUE si el argumento es una dirección externa
(**sequencep** <arg>): TRUE si es un valor multi-campo
(**stringp** <arg>): TRUE si el argumento es un string
(**symbolp** <arg>): TRUE si el argumento es un símbolo

Comandos para la gestión de variables multi-valuadas

Para encontrar un elemento en una lista se utiliza el comando `member$` :

```
(member$ <single-field-expression> <multifield expression>)
```

Ejemplos:

```
CLIPS> (member$ blue (create$ red 3 "text" 8.7 blue))
5
CLIPS> (member$ 4 (create$ red 3 "text" 8.7 blue))
FALSE
CLIPS>
```

Para obtener la longitud de una lista se utiliza el comando `length$`:

```
(length$ <expression>*)
```

Ejemplo:

```
(defrule paint-jug-in-red
 ?f <-  (jug name ?name $?z colours $?colours)
        (test (not (member$ red $?colours)))   ;; la jarra no está pintada de rojo
        (test (< (length$ $?colours) 2))     ;; la jarra  está pintada de un
                                            ;; color como mucho
=>
     (retract ?f)
     (assert (jug name ?name $?z colours $?colours red)))
```

Para crear una lista multivaluada se utiliza el comando `create$`:

```
(create$ <expression>*)
```

Ejemplos:

```
CLIPS> (create$ hammer drill saw screw pliers wrench)
(hammer drill saw screw pliers wrench)
CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))
```

```
(7 6 2)
CLIPS> (create$)
()
CLIPS> (bind ?lista (create$ 1 2 3 4))
(1 2 3 4)
```

Otro ejemplo:

```
(defrule paint-jug-in-red
    ?f <-  (jug name ?name $?z colours $?colours)
      (test (not (member red $?colours)))
=>
     (retract ?f)
     (assert (jug name ?name $?z colours (create$ $?colours red))))
```

Para recuperar el elemento de una lista en una posición determinada se utiliza el comando `nth$`: `(nth$ <integer-expression> <multifield-expression>)`

Ejemplos:

```
CLIPS> (nth$ 3 (create$ a b c d e f g))
c
CLIPS>
```

Para borrar un elemento de una lista se utiliza el comando `delete-member$`:
`(delete-member$ <multifield-expression> <expression>+)`

Ejemplos:

```
CLIPS> (delete-member$ (create$ 3 6 78 4 5 6 12 32 5 8 11 5 6)
(create$ 5 6))
(3 6 78 4 12 32 5 8 11)
```

```
CLIPS> (delete-member$ (create$ 3 6 78 4 5 6 12 32 5 8 11 5 6) 5 6)
(3 78 4 12 32 8 11)
CLIPS>
```

Para reemplazar un elemento de una lista se utiliza el comando `replace-member$`

```
(replace-member$  <multifield-expression>
                  <substitute-expression>
                  <search-expression>+)
```

Ejemplos:

```
CLIPS> (replace-member$ (create$ 3 6 78 4 5 6 12 32 5 8 11 5 6) HOLA
(create$ 5 6))
(3 6 78 4 HOLA 12 32 5 8 11 HOLA)
```

```
CLIPS> (replace-member$ (create$ 3 6 78 4 5 6 12 32 5 8 11 5 6) HOLA
5 6)
(3 HOLA 78 4 HOLA HOLA 12 32 HOLA 8 11 HOLA HOLA)
CLIPS>
```

Para insertar un elemento en una lista se utiliza el comando `insert$`:

```
(insert$ <multifield-expression>
         <integer-expression>
         <single-or-multi-field-expression>+)
```

Ejemplos

```
CLIPS> (insert$ (create$ a b c d) 1 x)
(x a b c d)
CLIPS> (insert$ (create$ a b c d) 4 y z)
(a b c y z d)
CLIPS> (insert$ (create$ a b c d) 5 (create$ q r))
(a b c d q r)
CLIPS>
```

Para encontrar el primer elemento de una lista se utiliza el comando `first$`:
`(first$ <multifield-expression>)`

Ejemplos

```
CLIPS> (first$ (create$ a b c))
(a)
CLIPS> (first$ (create$))
()
CLIPS>
```

Para encontrar el resto de elementos de una lista se utiliza el comando `rest$`:
`(rest$ <multifield-expression>)`

Ejemplos

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

Comando `read`

Para leer datos de un fichero: `(read [<logical-name>])`

Ejemplos:

```
CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" mydata)
TRUE
CLIPS> (read mydata)
red
CLIPS> (read mydata)
green
CLIPS> (read mydata)
EOF
CLIPS> (close)
TRUE
CLIPS>
```

Comando `readline`

Para leer una línea de un fichero: `(readline [<logical-name>])`

Ejemplos:

```
CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" mydata)
TRUE
CLIPS> (readline mydata)
"red green"
CLIPS> (readline mydata)
EOF
CLIPS> (close)
TRUE
CLIPS>
```