DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Dep. Sistemes Informàtics i Computació
Escola Tècnica Superior d'Enginyeria Informàtica
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

SISTEMES INTEL·LIGENTS

Seminari de CLIPS (V6.3)

(C Language Integrated Production System)

CLIPS V6.3

(C Language Integrated Production System)

1. Introducció

CLIPS és una eina de Sistemes Experts desenvolupada originalment per *Software Technology Branch* de la NASA/Lyndon B. Johnson Space Center. CLIPS està dissenyat per a la construcció de Sistemes Basats en Regles (SBR) i facilitar el desenvolupament de programari que requereix modelitzar coneixement d'experts en un determinat problema.

Hi ha tres formes de representar coneixement en CLIPS:

- **regles**: especialment destinades per a representar coneixement heurístic basat en l'experiència
- **funcions**: per a representar coneixement procedural
- **programació orientada a objectes (POO)**: per a representar principalment coneixement procedural. CLIPS suporta les 6 característiques generalment acceptades de la POO: classes, pas de missatges, abstracció, encapsulament, herència i polimorfisme.

CLIPS suporta per tant els 3 paradigmes de programació:

- **programació basada en regles**: regles + fets (facts)
- **programació procedural:** funcions
- **programació orientada a objectes**: objectes + pas de missatges

Dels tres paradigmes, nosaltres solament utilitzarem la programació basada en regles i, puntualment, la programació procedural.

Característiques de CLIPS:

- CLIPS és una eina escrita en C.
- Plena integració amb altres llenguatges com a C i ADA.
- Des d'un llenguatge procedural es pot cridar a un procés CLIPS; aquest realitza la seua funció i després li retorna el control al programa.
- Es pot incorporar codi procedural com a funcions externes en CLIPS.
- Les regles poden fer "pattern-matching" sobre objectes i fets formant així un sistema integrat.
- CLIPS té una sintaxi estil LISP que utilitza parèntesi com a delimitadors.

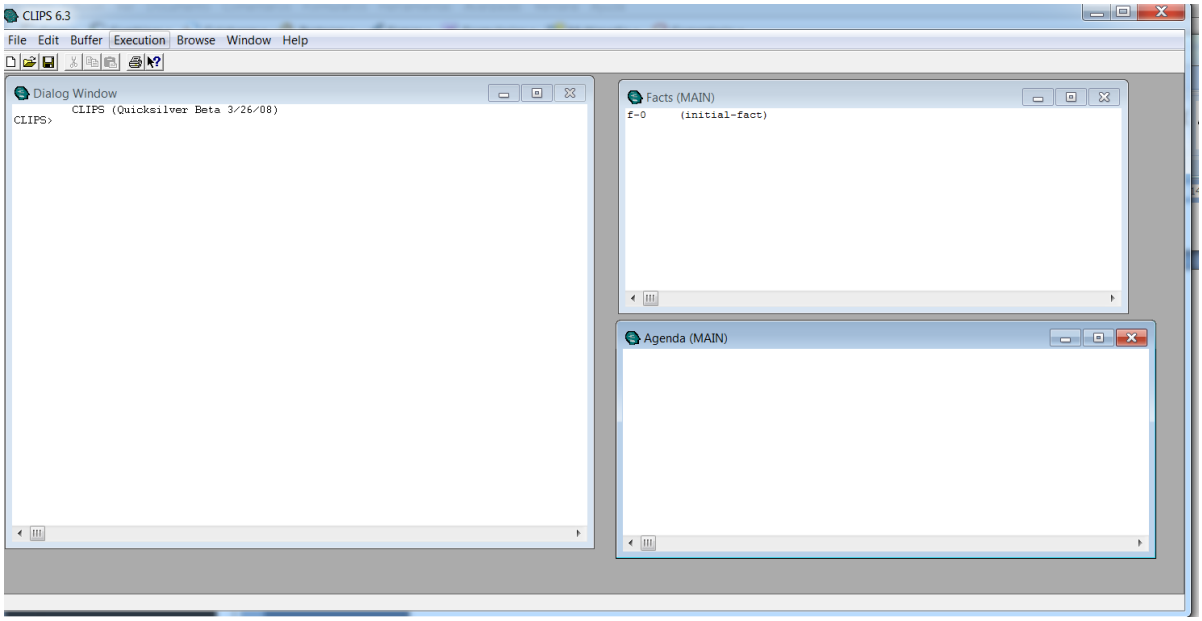
CLIPS és un Sistema de Producció que es compon dels següents mòduls:

- Memòria Global o **Base de Fets** (fets + instàncies d'objectes)
- **Base de Regles** o Base de Coneixements (regles)
- **Motor d'Inferència** (control)

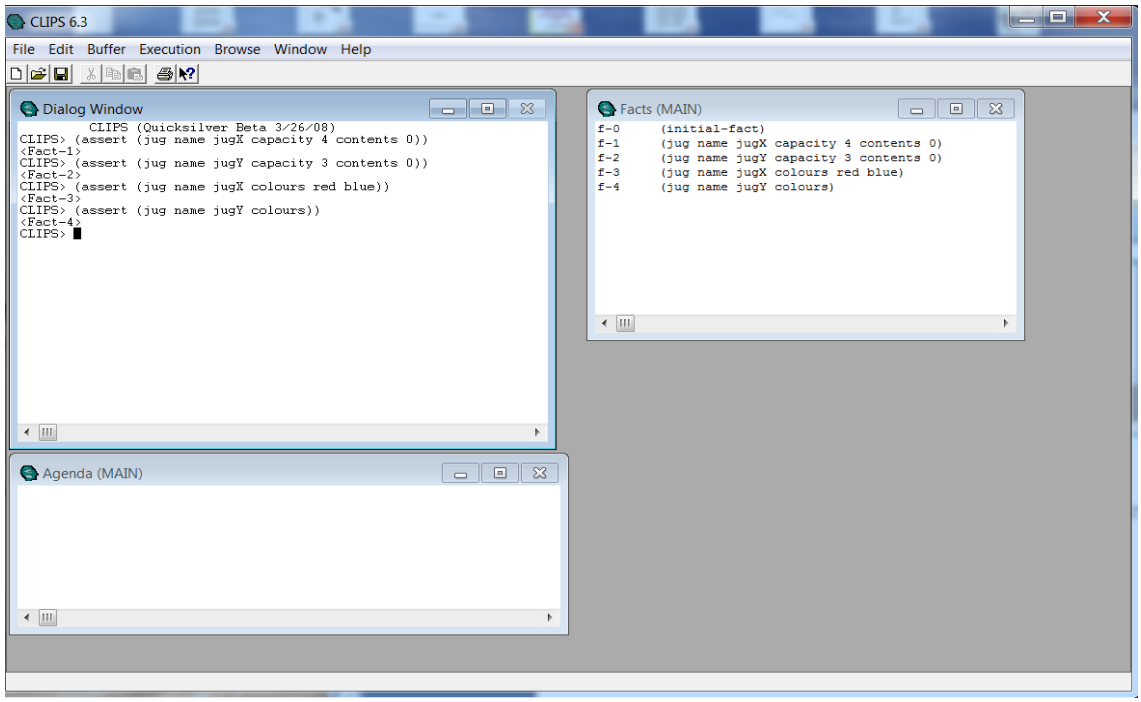
2. Interfície de CLIPS

La interfície de CLIPS es compon de tres finestres principals:

- **Dialog Window**. Intèrpret de CLIPS o finestra del símbol del sistema on es pot teclejar comandos executables que CLIPS avaluarà.
- **Facts**. Base de Fets que conté els fets del problema
- **Agenda.** Agenda o Conjunt conflicte on s'emmagatzemaran les instàncies de regles o activacions.



Exemples d'utilització del comando `assert`.

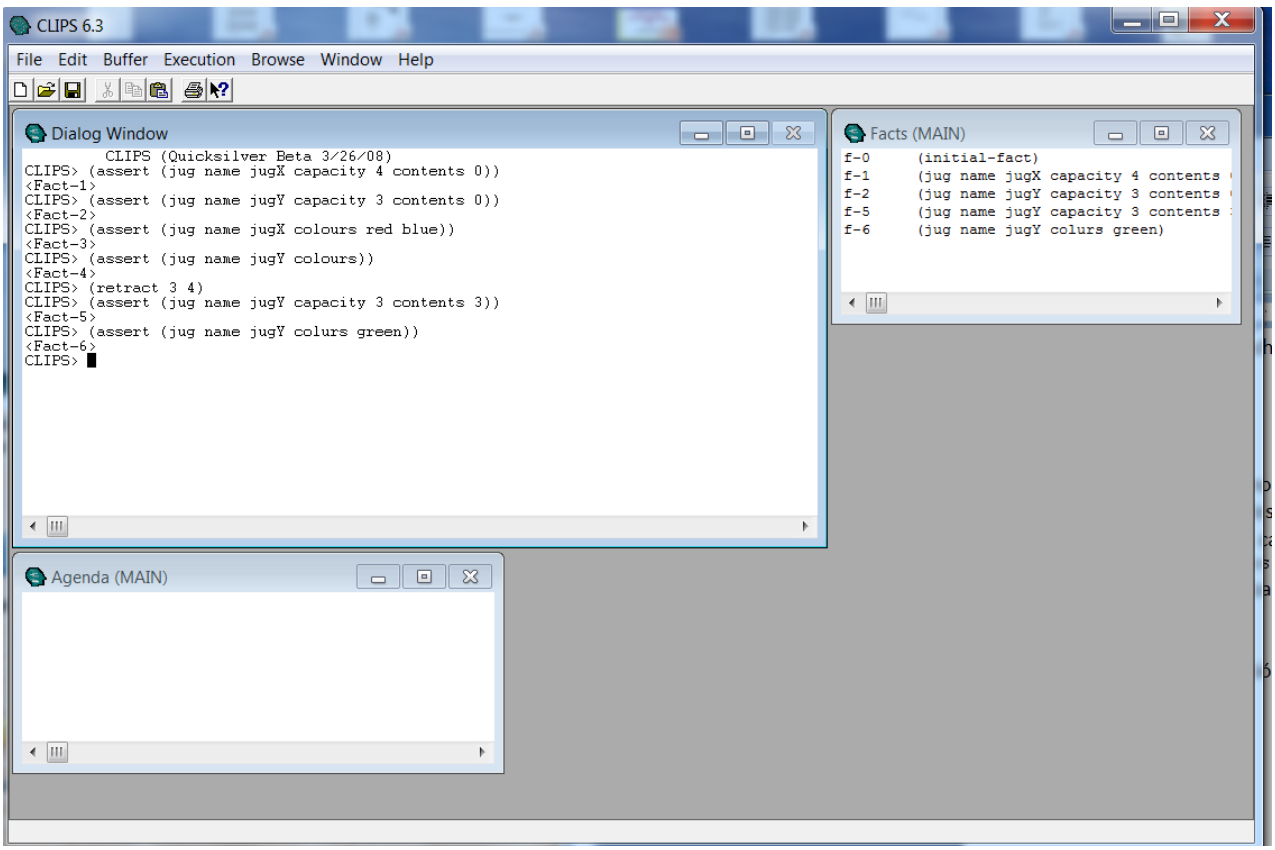


- El comando **`assert`** s'utilitza per a inserir un fet en la Base de Fets (BF). La sintaxi és `(assert <fact>+)`.
- Els fets representen un conjunt de variables, propietats i valors que aquestes poden prendre.
- Un fet es compon d'un nombre il·limitat de camps separats per blancs i tancats entre parèntesis balancejats. Els camps poden tenir o no un nom assignat.
- Tots els fets tenen un identificador o índex de la forma **f-n** (veure finestra **Facts**). on '**n**' és l'índex del fet (`fact-index`).
- CLIPS defineix per defecte un fet inicial `(initial-fact)` quan es carrega l'aplicació.
- Tots els fets s'insereixen en la llista ***Facts Window*** (BASE DE FETS).

Fets ordenats:

- Els camps no tenen nom assignat
- L'ordre dels camps és significatiu
- Tipos dels camps: `float`, `integer`, `symbol`, `string`, `external-address`, `fact-address`, `instance-name`, `instance-address`
- Se sol utilitzar el primer camp d'un fet per a descriure una relació entre camps.

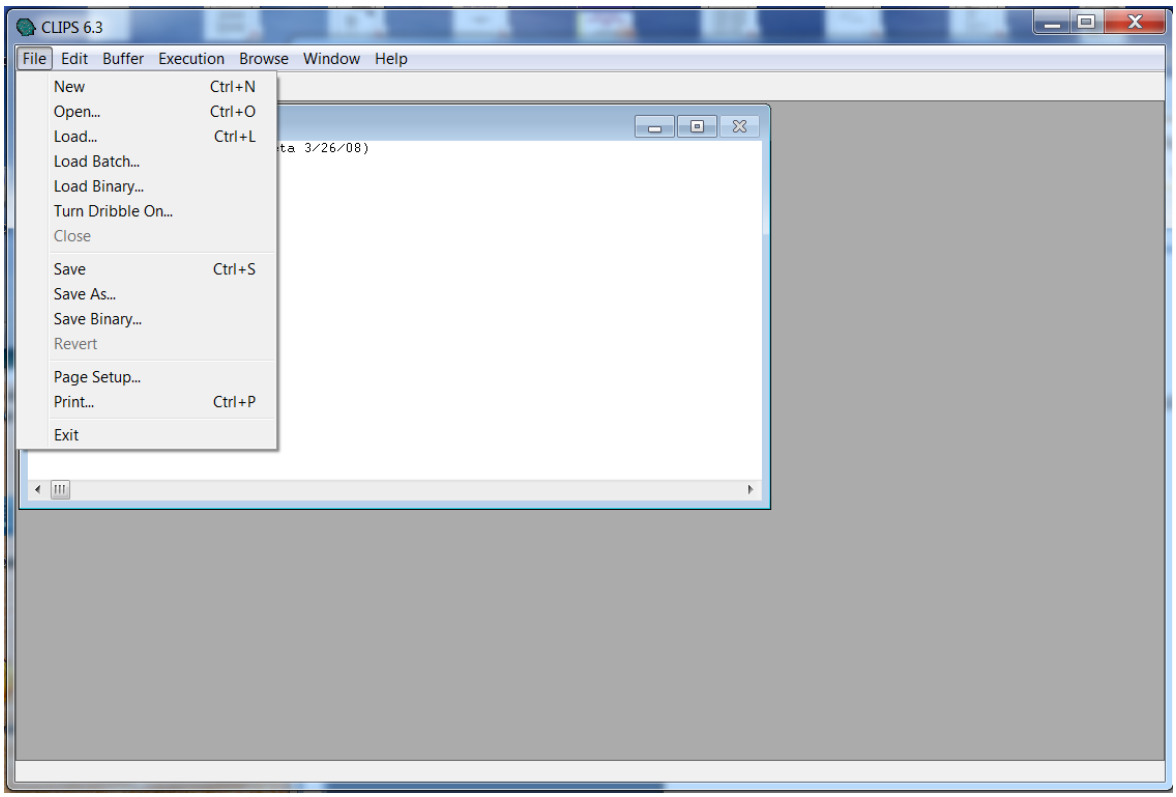
Exemples d'utilització del comando `retract`.



- El comando `retract` s'utilitza per a eliminar un fet de la BF. La sintaxi d'aquest comando és `(retract <fact-index>+)`.
- Quan s'elimina un fet, la resta conserva el seu índex o identificador original
- Qualsevol nou fet que s'inserisca en la BF se li assigna l'índex següent al de l'últim fet inserit

2.1 Entorn CLIPS

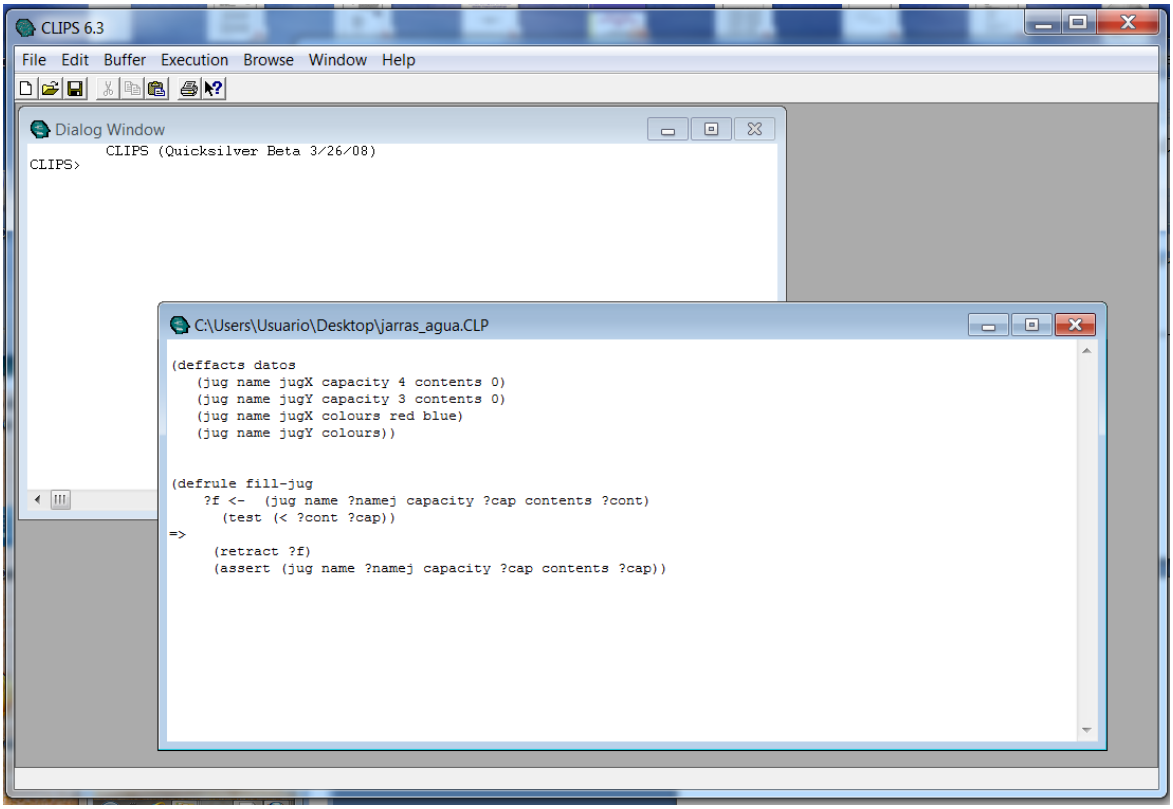
Escriu el teu SBR en un fitxer de text. Pots utilitzar qualsevol editor de text o bé l'editor de CLIPS (opció **New** del menú **File**).



A continuació es mostra un exemple d'un SBR que solament conté un conjunt de fets inicials, definits mitjançant el constructor `deffacts` i una regla definida mitjançant el constructor `defrule`.

Mitjançant el constructor `deffacts` es defineixen tots els fet inicials del problema. El constructor `deffacts` simplement emmagatzema en memòria l'estructura definida per a introduir els fets en el moment d'executar el SBR. La sintaxi és

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



Les tres fases per a engegar un SBR són:

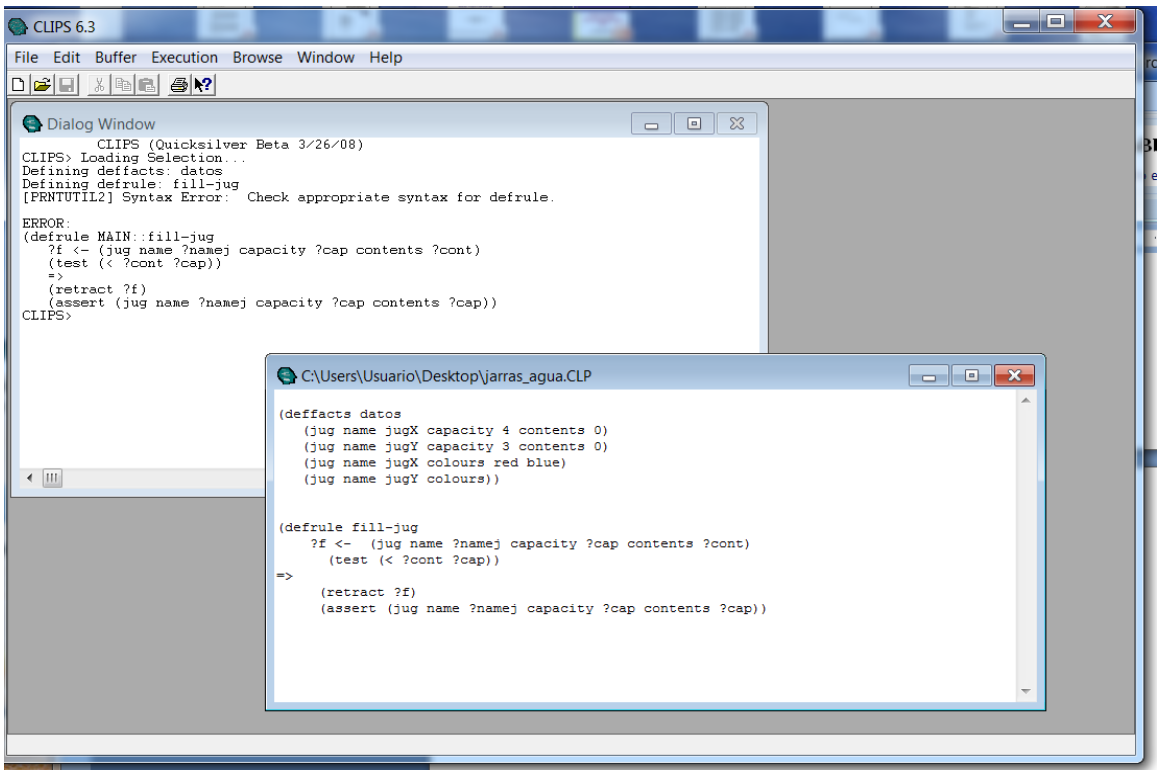
- **Carregar el fitxer**. Aquesta operació realitza un parsing del fitxer per a comprovar que no hi ha errors sintàctics.
- **Reset CLIPS**. Buida el contingut actual de la BF i l'agenda; carrega els fets definits en l'estructura deffacts, les regles definides mitjançant defule i realitza la primera fase de matching.
- **Executar el SBR**. Aquesta operació engega el Motor d'Inferència i executa el SBR, aplicant successivament el cicle reconeixement-acció de l'algorisme RETE.

Carregar fitxer

Després de guardar el fitxer, la següent operació és carregar el fitxer. Açò es pot fer de dues formes:

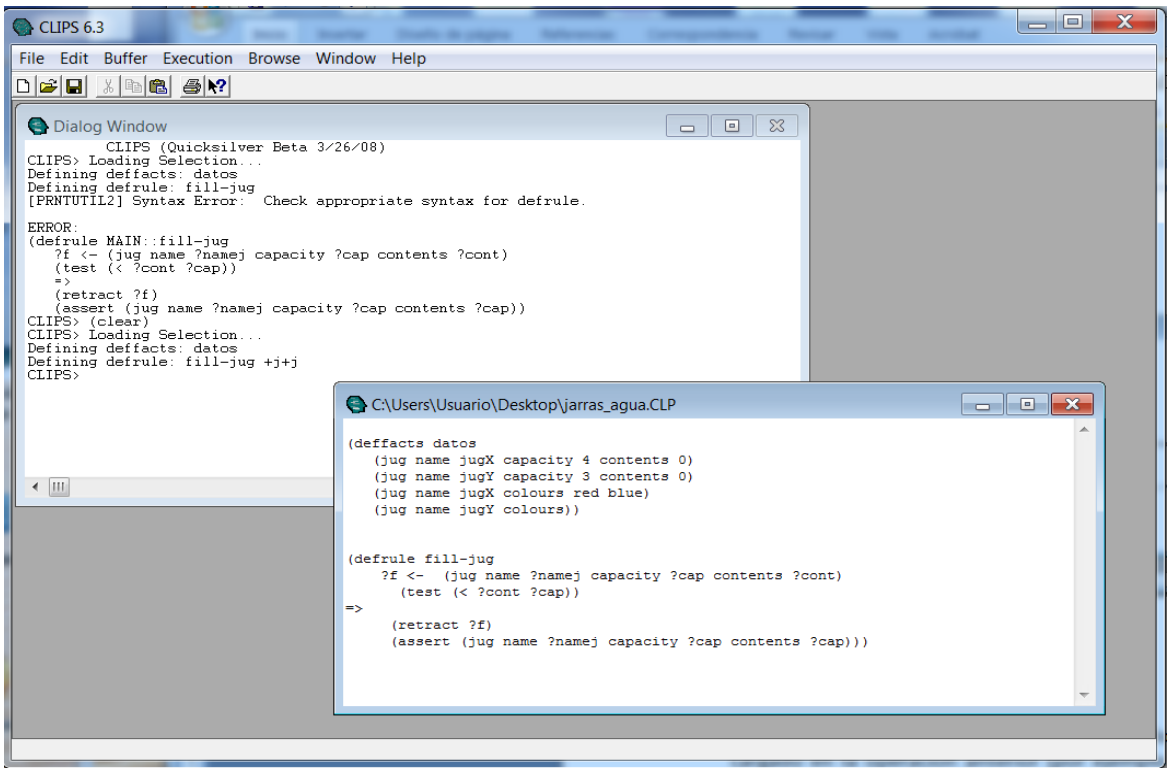
- Utilitzant l'opció **Load …** del menú **File**
- Utilitzant l'opció **Load Buffer** del menú **Buffer**

Aquesta operació comprova que totes les estructures CLIPS del fitxer estan correctament escrites. En l'exemple que es mostra a continuació, es pot observar que hi ha un error en la definició de la regla; concretament, l'error se situa al final de la regla, on falta un parèntesi de tancament.



En aquest cas, el que ha de fer-se és:

a. Corregir l'error del fitxer
b. Teclejar el comando `(clear)` per a esborrar totes les estructures que pogueren haver-se carregat en l'operació anterior (per exemple, l'estructura `deffacts`). El comando `(clear)` també pot activar-se des de l'opció **Clear CLIPS** del menú **Execution**
c. Tornar a carregar el fitxer



Reset CLIPS

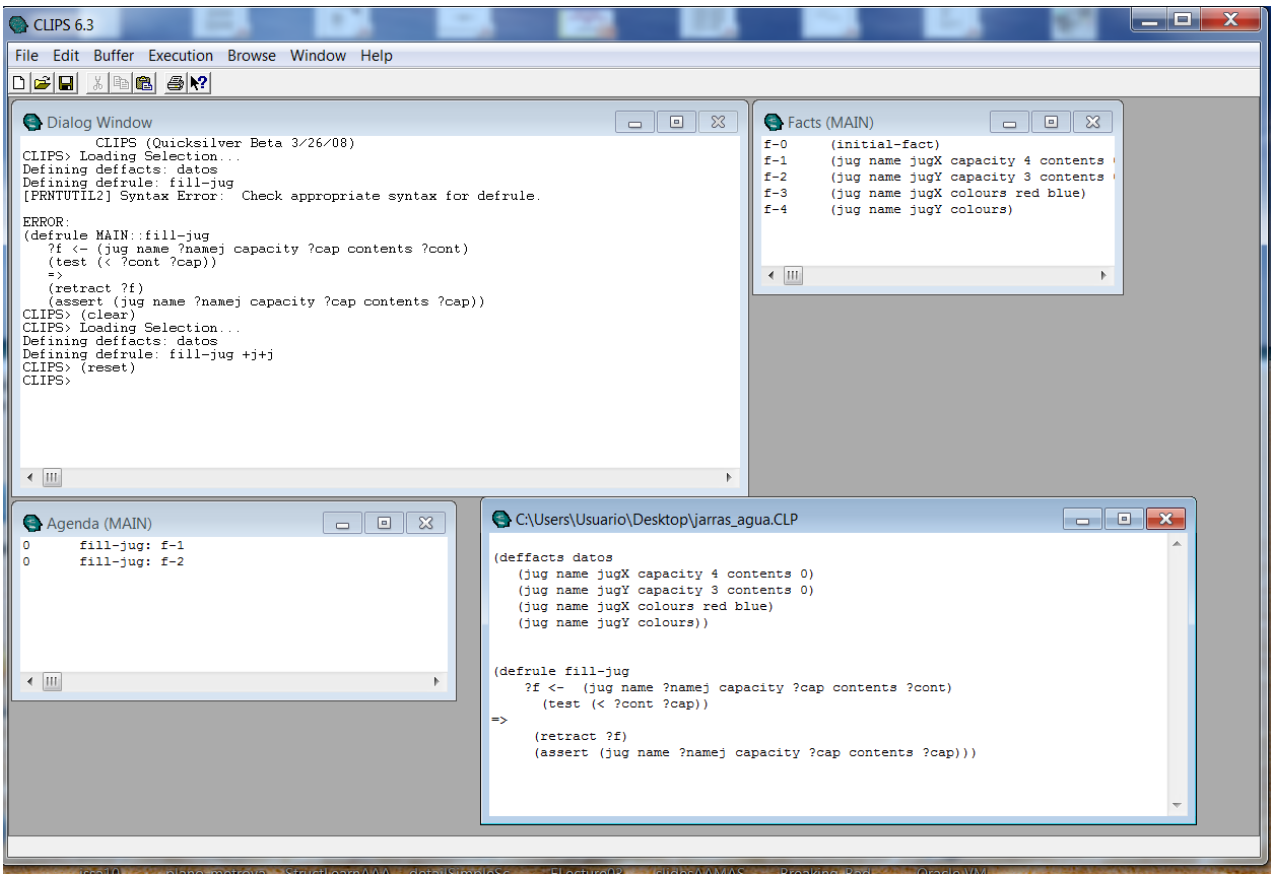
Una vegada que el fitxer està lliure d'errors, la següent operació és carregar les estructures de dades en memòria. Concretament, el comando `(reset)` realitza les següents operacions:

1. Esborra els fets existents en la BF de l'execució anterior.
2. Insereix el fet inicial (initial-fact).
3. Insereix els fets definits en els constructors `deffacts` en la BF
4. Realitza la primera fase de matching construint les possibles activacions de regles que puguen existir.

El comando `(reset)` pot activar-se:

a. Directament en la finestra de comandos (Dialog Window)
b. Mitjançant l'opció **Reset** del menú **Execution**

Com pot observar-se en la següent pantalla, l'operació `(reset)` ha introduït els fets en la BF i dues activacions en l'Agenda (la regla `fill-jug` per al pitxer de nom `jugX`, i la mateixa regla `fill-jug` per al pitxer de nom `jugY`).



Executar CLIPS

El tercer i últim pas és executar el SBR. Aquesta operació pot realitzar-se de dues formes:

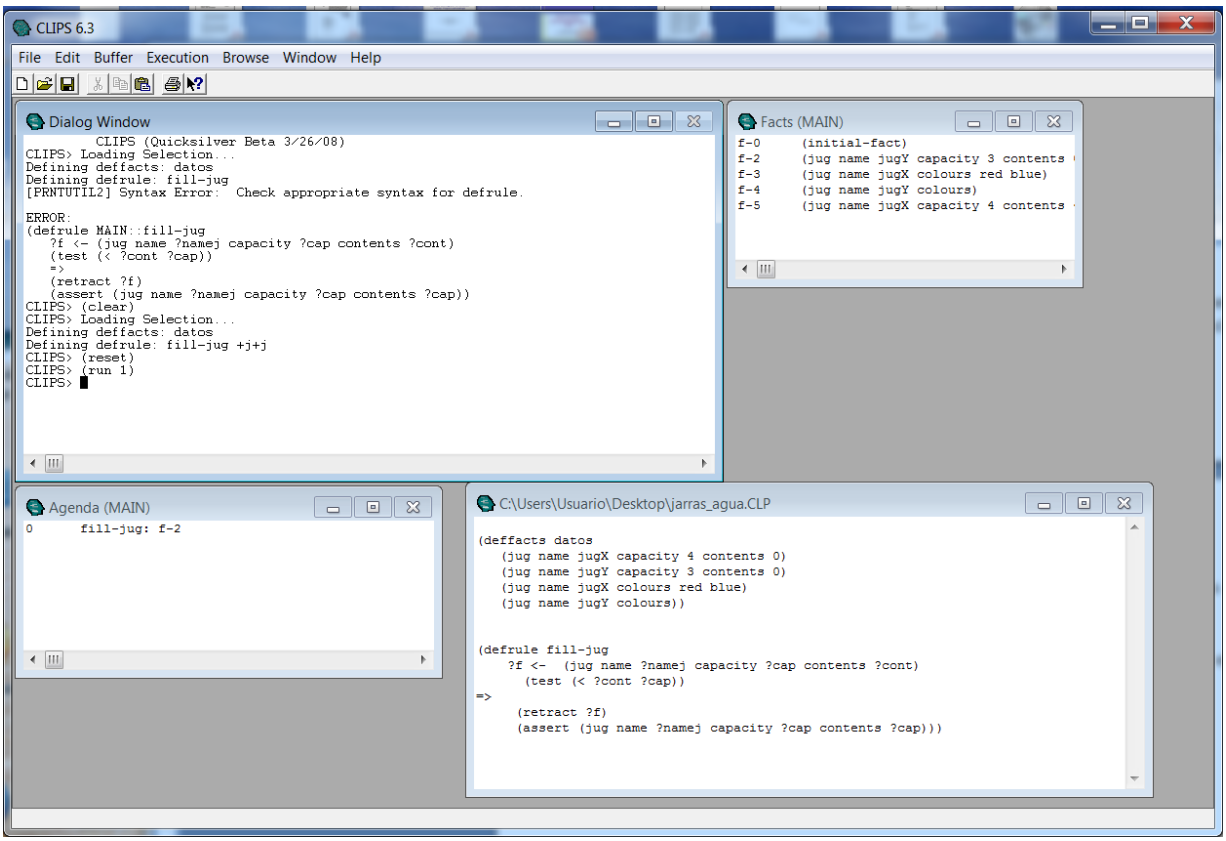
a. Teclejant el comando `(run)` o seleccionant l'opció **Run** del menú **Execution**. El comando Run executa successivament el cicle match-selecció-execució fins que el sistema finalitza (o bé una regla per al procés inferencial, o bé l'agenda es queda buida)

b. Mitjançant l'opció **Step** del menú **Execution.** Aquesta opció permet anar executant el SBR pas a pas, veient cada cicle del procés inferencial.

Seleccionem l'opció **Step** en el nostre problema. Cada operació Step realitza les següents fases:

1. selecciona la primera activació de l'Agenda
2. executa la RHS de la instància seleccionada
3. realitza de nou el procés de matching.

En el nostre xicotet exemple, l'execució de la primera instància de l'agenda esborra el fet **f-1** i genera el fet **f-5** (que representa que el pitxer jugX està ara ple) però el fet **f-5** no provoca cap nou matching perquè no activa l'única regla que tenim definida en el SBR.



A continuació tornaríem a executar **Step** del menú **Execution**, se seleccionaria l'única instància disponible en l'Agenda, s'executaria la seua RHS i generaria un nou fet **f-6** que no instancia la

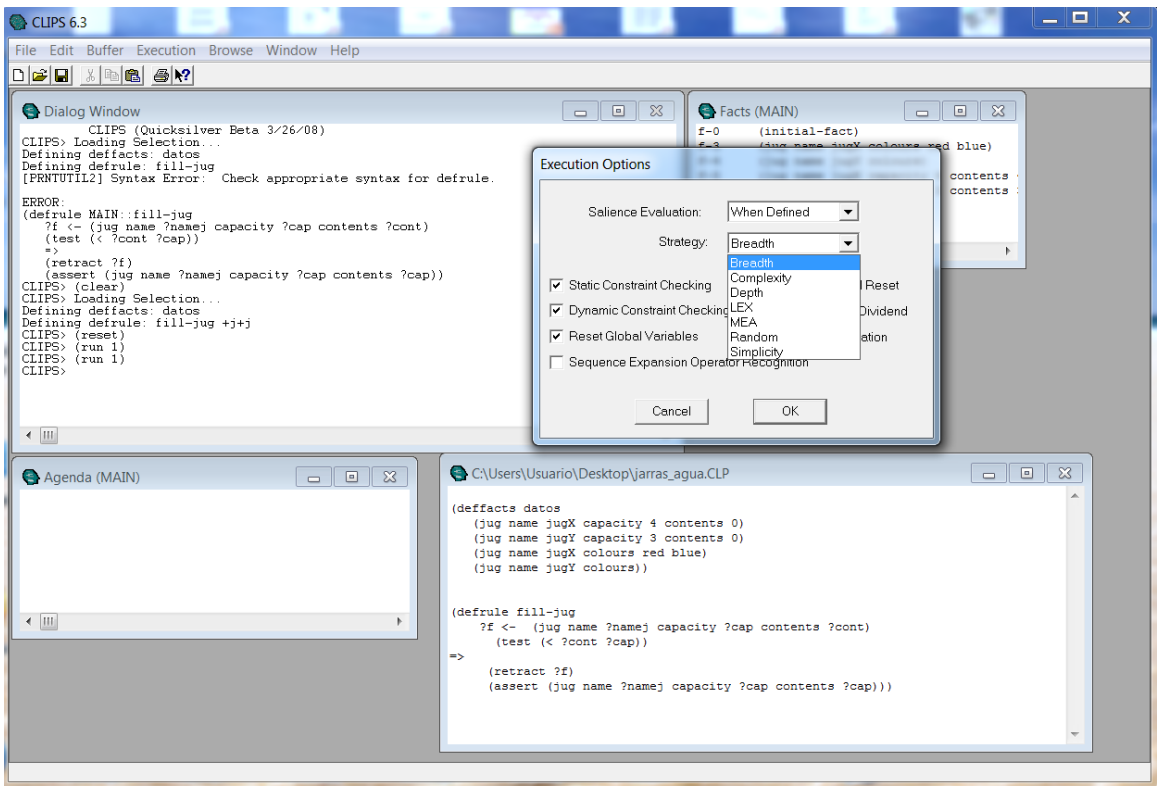
regla `fill-jug`. Per tant, la fase de matching no genera noves activacions. El procés inferencial es pararia llavors en aquest punt.

2.2 Estratègies de Control

Per a una definició detallada de les ***estratègies de resolució de conflictes*** en CLIPS (refracció, no duplicitat de fets, etc.), veure transparències del tema 7.

Bàsicament, en els nostres exemples treballarem amb les següents estratègies de control: Amplària (**Breadth**), Profunditat (**Depth**) i, eventualment, una estratègia aleatòria (**Random**).

Per a seleccionar una estratègia de control en CLIPS, cal anar a l'opció **Options** del menú **Execution**. En aquesta nova pantalla, punxar en el menú desplegable de **Strategy** i ací trobarem les opcions d'estratègies de control disponibles en CLIPS.

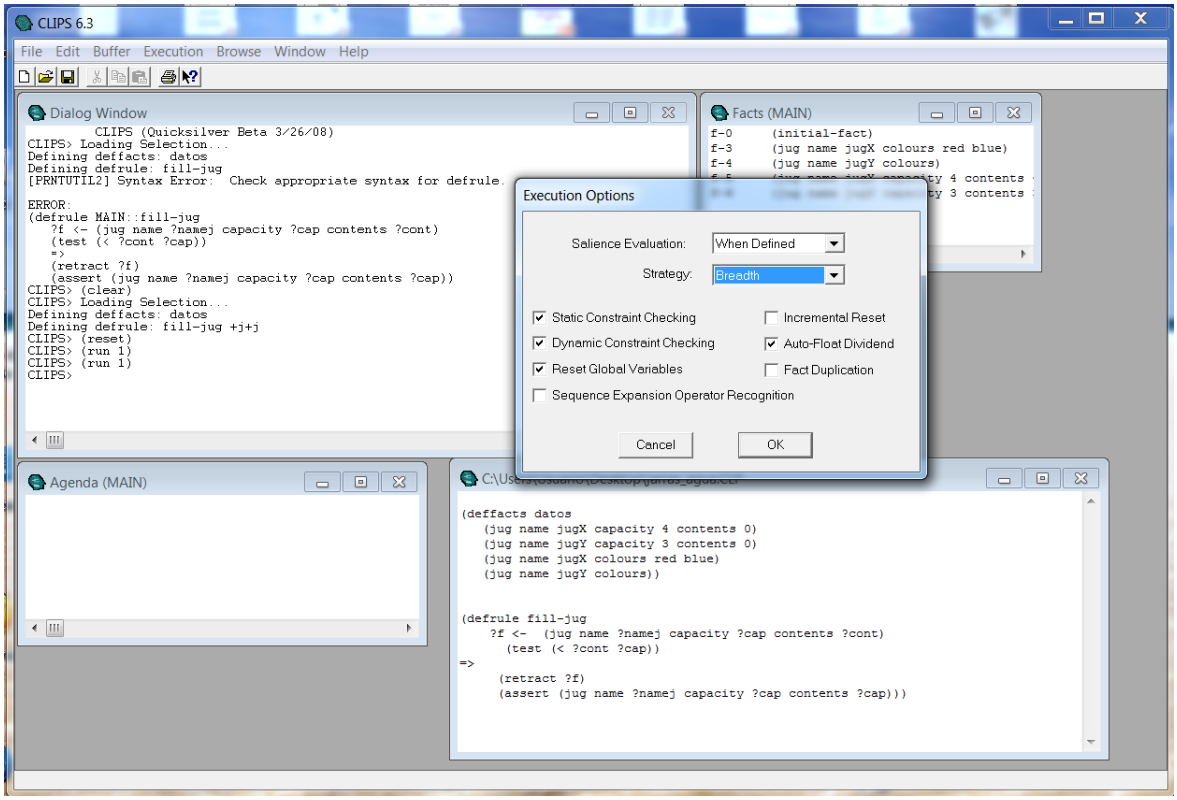


Així mateix, pot observar-se que en la pantalla d'Execution **Options** apareix l'etiqueta **Fact Duplication** sense marcar (opció per defecte de CLIPS).

Recordatoris:

1. CLIPS utilitza refracció: no permet activar una regla més d'una vegada amb els mateixos fets (fets amb els mateixos nombres d'índex).
2. CLIPS per defecte no permet Fact Duplication

Per a més detalls, veure transparències del Tema 7.



3. Motor d'inferència de CLIPS

En aquest apartat, es presenten dos exemples de SBR per a analitzar el comportament del cicle reconeixement-acció (o cicle matching-selecció-execució) de l'algorisme RETE.

3.1 Ordenació d'una llista de nombres

Siga una BF que conté un fet que representa una llista de nombres naturals no ordenats. Escriure una única (si és possible) regla de producció que obtinga, com BF final, la llista inicial amb els seus elements en ordre creixent:

Exemple: (llista 4 5 3 46 12 10) obté: (llista 3 4 5 10 12 46)

El fitxer CLIPS seria el següent (es pot descarregar aquest codi del fitxer OrdenarNumeros.clp).

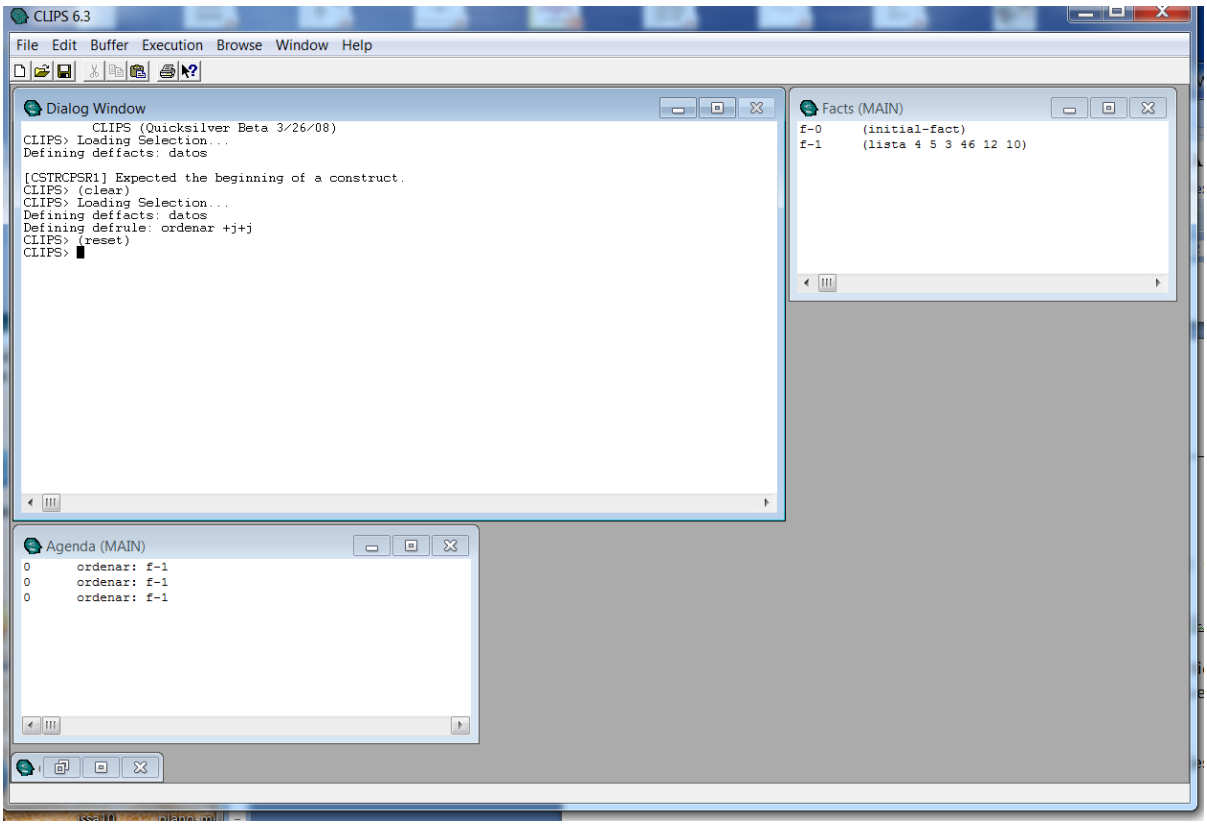
```
(deffacts dades
     (llista 4 5 3 46 12 10))

(defrule ordenar
   ?f1 <- (llista $?x ?y ?z $?w)
          (test (< ?z ?y))    ;; comprovem si ?z és menor que ?y
  =>
      (retract ?f1)
      (assert (llista $?x ?z ?y $?w)))  ;; intercanviem elements
```

En aquest exemple, quan el Motor d'Inferència pare serà perquè no té més instàncies de la regla `ordenar` en l'agenda, la qual cosa significa que ja estaran tots els elements de la llista original ordenats.

Seleccionem **Breadth** (Amplària) com a estratègia de resolució de conflictes (aquest problema es correspon amb l'exemple 2 del tema 7).

Després de carregar el fitxer, i teclejar `(reset)`, obtenim el següent resultat:



Es pot observar que en l'Agenda hi ha tres activacions de la regla `ordenar`. Totes elles instancien amb l'únic fet disponible en la BF:

1. La primera instància correspon als intercanvis dels valors 12 10.
2. La segona instància correspon als intercanvis dels valors 46 12.
3. La tercera instància correspon a l'intercanvi dels valors 5 3.

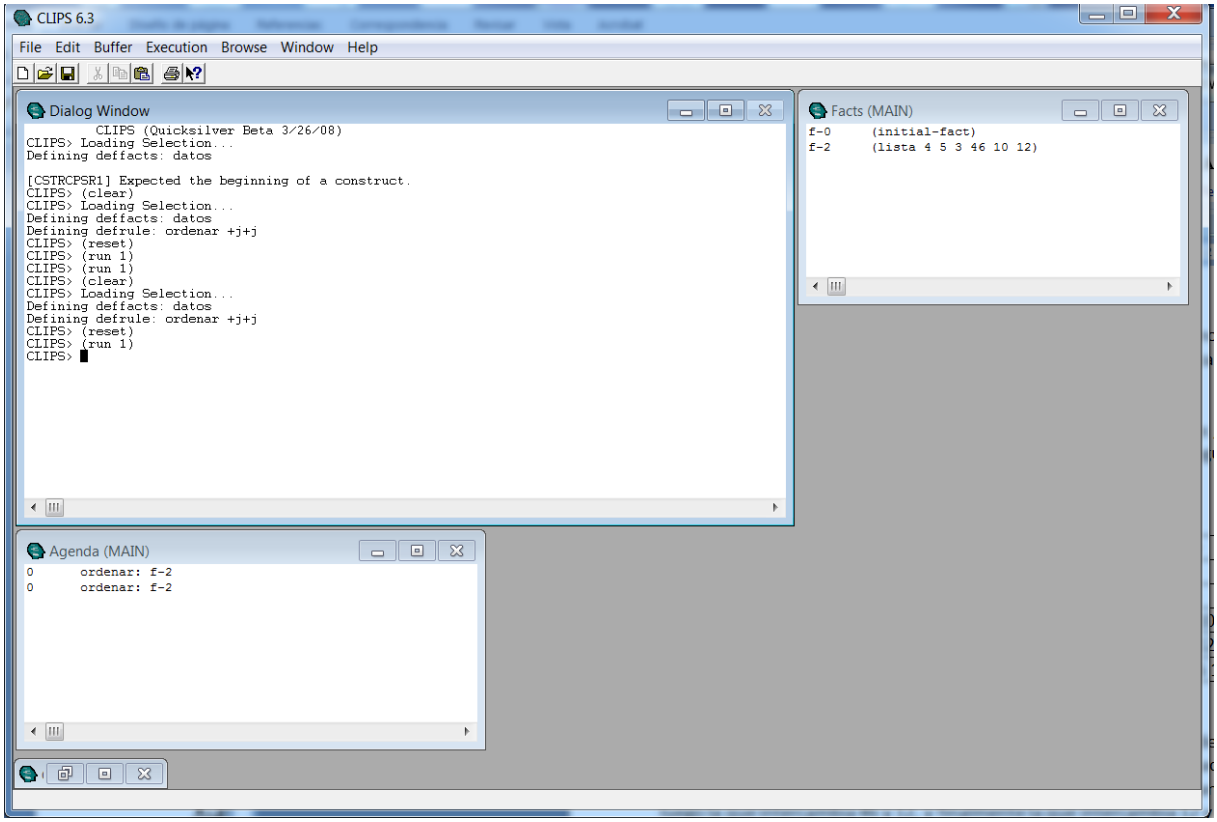
L'ordre en el qual l'estratègia d'Amplària insereix les activacions trobades en la fase de matching d'un cicle té a veure amb l'ordre en el qual CLIPS realitza el procés de pattern-matching. CLIPS comença per l'última regla del SBR, últim patró de la LHS d'una regla, o última variable multi-valuada d'un patró d'una regla.

D'aquesta manera, CLIPS intenta primer lligar zero elements a la variable multi-valuada `$?w` (o, equivalentment, tots els possibles elements a la variable `$?x`); a continuació lliga un sol element a la variable `$?w`, després dos elements, i així successivament. Concretament, l'ordre en el qual CLIPS fa el pattern-matching és:

#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

L'estratègia d'AMPLÀRIA introdueix les activacions en el mateix ordre en el qual CLIPS realitza el pattern-matching perquè afegeix cada nova activació al final. En canvi, l'estratègia de PROFUNDITAT introduiria les activacions en l'ordre invers al mostrat en la taula perquè afegeix cada nova activació al principi; és a dir, primer introduiria la instància que intercanvia 5 i 3, després la que intercanvia 46 i 12, i finalment la que intercanvia 12 i 10.

Seguint amb l'estratègia en AMPLÀRIA … executem el primer cicle `(step)` i el resultat és:



- S'elimina el fet f-1; com a conseqüència d'açò, s'eliminen també de l'Agenda les altres dues instància que depenien de f-1
- Es genera un nou fet f-2 on els elements 12 i 10 ja estan intercanviats
- Automàticament, es duu a terme la següent fase de matching i apareixen dues activacions amb el fet f-2 (la primera correspon als elements 46 i 10, i la segona als elements 5 i 3).

Continua l'execució del SBR pas a pas fins que l'Agenda es queda buida i comprova l'estat final de la BF.

3.2 Seqüències d'ADN

Donades dues seqüències d'ADN, escriu un SBR (una única regla, si és possible) que conte el nombre de mutacions (es pot utilitzar una variable global per a emmagatzemar el nombre d'encerts o bé un fet per a registrar aquest valor). Per exemple, per a les dues seqüències d'ADN

(A A C C T C G A A A) i (A G G C T A G A A A) hi ha tres mutacions.

El fitxer CLIPS seria el següent (es pot descarregar aquest codi del fitxer MutacionsADN.clp).

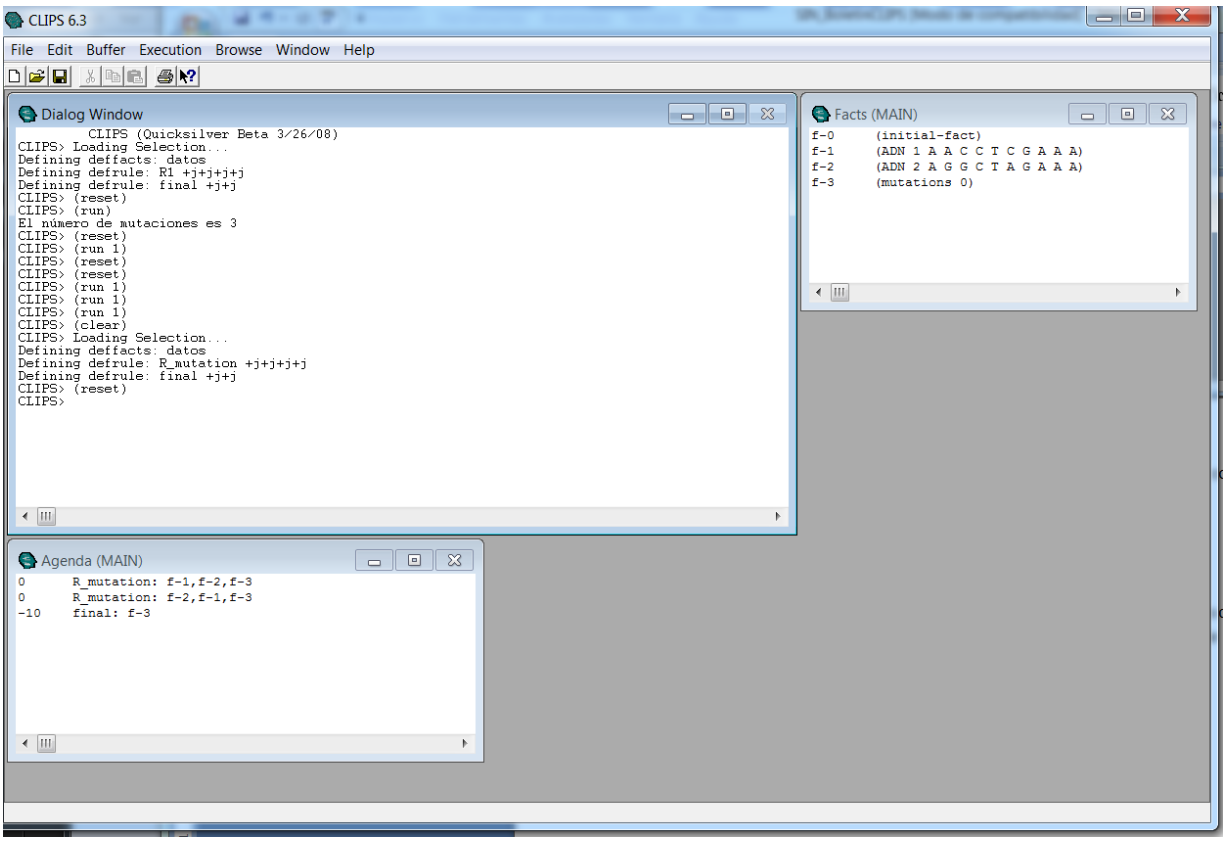
En aquest cas, hem definit una regla `final` que mostra per pantalla el resultat. Li donem a la regla final una prioritat menor perquè es llance únicament quan no hi haja activacions de la regla `R_mutation`).

```
(deffacts dades
     (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 ?y1 $?i1)
   ?f2 <-    (ADN ?n2 $?x ?y2 $?i2)
   ?f3 <-    (mutations ?m)
      (test (and (neq ?n1 ?n2)(neq ?y1 ?y2)))
   =>
      (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 nombre de mutacions és " ?m crlf))
```

Seleccionem **Depth** (Profunditat) com a estratègia de resolució de conflictes. Després de carregar el fitxer, i teclejar `(reset)`, obtenim el següent resultat:



Les dues instàncies que apareixen en l'Agenda es corresponen realment amb la mateixa mutació; açò es deu al fet que:

- f-1, f-2,f-3 fan matching amb la regla R_mutation, i
- f-2 ,f-1,f-3 també fan matching amb la regla R_mutation

No obstant açò, quan una activació es llance i execute la seua RHS, esborrarà els fets que han instanciat els patrons eliminant automàticament l'altra instància depenent dels mateixos fets.

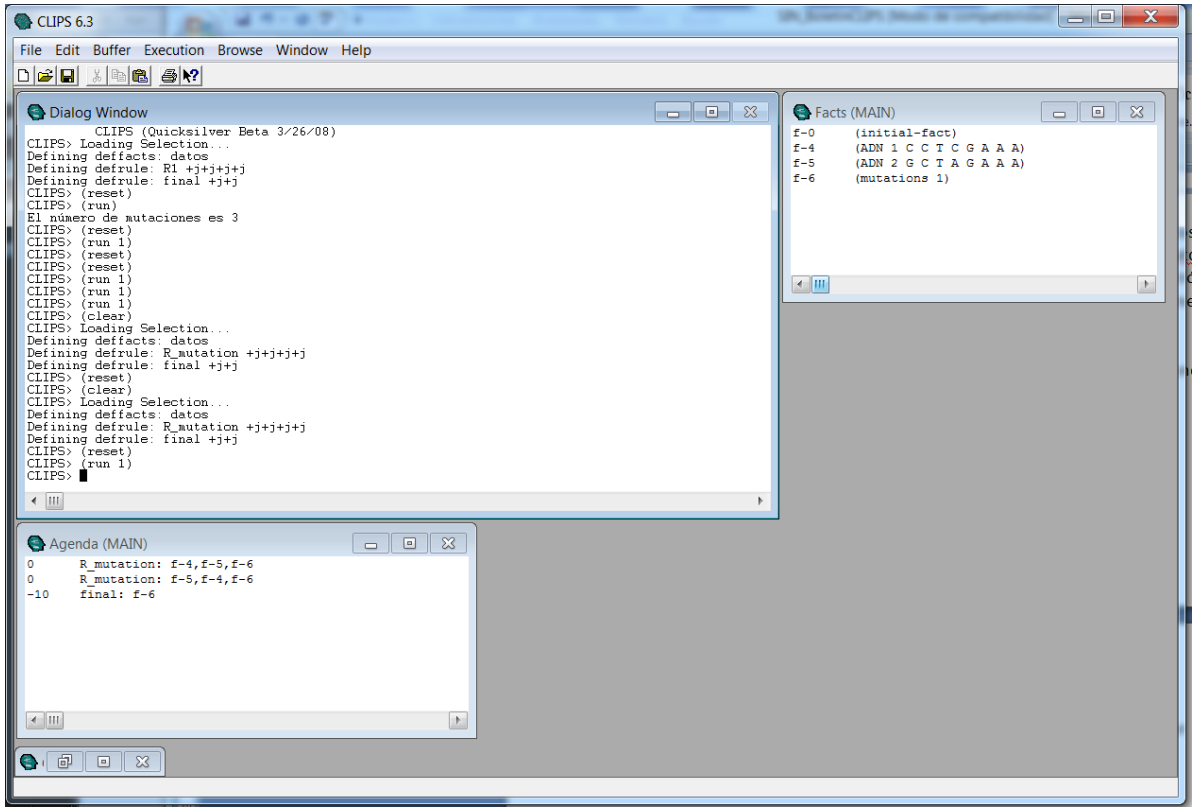
Com s'ha explicat anteriorment, l'ordre en el qual s'introdueixen les dues activacions en l'Agenda depèn de l'ordre en el qual CLIPS realitza el procés de pattern-matching i de l'estratègia de resolució seleccionada:

1. CLIPS comença pel tercer patró, instanciant-lo amb f-3; després instancia el segon patró amb el primer fet que troba f-1; conseqüentment, el primer patró s'instancia amb el fet f-2 perquè el test siga TRUE.
2. A continuació intenta cercar una altra combinació de fets que instancie la LHS de la regla: tercer patró amb f-3, segon patró amb el següent fet que troba, f-2, i, conseqüentment, el primer patró amb el fet f-1 perquè el test siga TRUE.

Les dues activacions que apareixen en l'Agenda, i que en realitat corresponen a la mateixa mutació, són:

#Match	**?n1**	**?n2**	**$?x**	**?y1**	**?y2**
1	1	2	(A)	A	G
2	2	1	(A)	G	A

Seguint amb l'estratègia en PROFUNDITAT … executem el primer cicle `(step)` i el resultat és:



- S'eliminen els fets f-1, f-2 i f-3; com a conseqüència d'açò, s'elimina també de l'Agenda l'altra activació
- Es generen tres nous fets f-4, f-5 i f-6
- Automàticament, es duu a terme la següent fase de matching i apareixen dues activacions corresponents a la mutació dels primers elements de cada seqüència d'ADN (C de la seqüència 1 i G de la seqüència 2).

Continua l'execució del SBR pas a pas fins que es dispare la regla final i mostre per pantalla el nombre total de mutacions. Pots provar aquest mateix SBR amb altres dues seqüències d'ADN.

ANNEX: Aspectes addicionals del llenguatge CLIPS

Els principals elements del llenguatge CLIPS s'han vist i utilitzat en les transparències corresponents als temes 6 i 7 del bloc 3 Representació del Coneixement. En aquest apartat presentem algunes elements addicionals de CLIPS que poden ser d'utilitat per al disseny i desenvolupament d'un SBR.

Comando `printout`

Aquest comando permet obtenir una eixida en el dispositiu representat per `<logical-name>`.

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

El dispositiu d'eixida pot ser un fitxer o bé la pantalla. El símbol **t** és el que s'utilitza per a identificar l'eixida estàndard. Exemples:

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

`crlf` significa *carriage return line feed* (tornada de carro)

Altre exemple d'utilització del comando `printout` en la RHS d'una regla per a mostrar resultats per pantalla.

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

Comando `bind`

Aquest és el comando que s'utilitza en CLIPS per a lligar un valor a una variable, monovaluada o multivaluada: `(bind <variable> <expression>*)`

Exemples:

```
(defrule roll-the-diu
      (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 globals

Les variables globals han de definir-se al principi del programa mitjançant el comando **`defglobal`**. El nom d'una variable global sempre ha d'anar tancat entre dues asteriscs (*).

Exemple de declaració de variables globals:

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

Es pot lligar un valor a una variable al mateix temps que es defineix dins del comando defglobal.

Un altre exemple:

```
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`

Com en altres llenguatges, CLIPS permet definir funcions pròpies mitjançant la primitiva **`deffunction`**. Les funcions definides amb `deffunction` tenen un àmbit global i són cridades

de la mateixa manera que qualsevol altra funció. A més aquestes funcions es poden utilitzar com a arguments d'altres funcions.

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

1) $?arg_i$: paràmetres
2) la funció només retorna el valor de l'última acció <action k>. Aquesta acció pot ser una funció, una variable o una constant.

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


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

Programació procedural

Es pot incloure codi procedural en la part dreta d'una regla. Aquestes estructures són:

```
- while
- if then else
- break
- return
```

Exemple:

```
(deffunction inici ()
   (reset)
   (printout t "Profunditat Maxima:= " )
   (bind ?*prof* (read))
   (printout t "Tipus de Cerca " crlf "
               1.- Amplària" crlf "    2.- Profunditat" crlf )
   (bind ?a (read))
   (if (= ?a 1)
       then   (set-strategy breadth)
       else   (set-strategy depth))
  (printout t " Executa run per a engegar el programa " crlf))
```

Funcions de predicat

Funcions de predicat que es poden usar en els tests condicionals de les regles.

(**evenp** <arg>): TRUE si el nombre és parell
(**floatp** <arg>): TRUE si és un nombre en coma flotant
(**integerp** <arg>): TRUE si és un nombre enter
(**lexemep** <arg>): TRUE si l'argument és un símbol o una cadena
(**numberp** <arg>): TRUE si l'argument és un nombre
(**oddp** <arg>): TRUE si el nombre és imparell
(**pointerp** <arg>): TRUE si l'argument és una adreça externa
(**sequencep** <arg>): TRUE si és un valor multi-camp
(**stringp** <arg>): TRUE si l'argument és una cadena
(**symbolp** <arg>): TRUE si l'argument és un símbol

Comandos per a la gestió de variables multi-valuadas

Per a trobar un element en una llista s'utilitza el comando `member$` :

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

Exemples:

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

Per a obtenir la longitud d'una llista s'utilitza el comando `length$`:

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

Exemple:

```
(defrule paint-jug-in-xarxa
 ?f <-  (jug name ?name $?z colours $?colours)
        (test (not (member$ red $?colours)))    ;; el pitxer no està pintat de roig
        (test (< (length$ $?colours) 2))        ;; el pitxer està pintat d'un
                                                ;; color com a molt
=>
     (retract ?f)
     (assert (jug name ?name $?z colours $?colours red)))
```

Per a crear una llista multivaluada s'utilitza el comando `create$`:

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

Exemples:

```
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$)
()
CLIPS> (bind ?llista (create$ 1 2 3 4))
(1 2 3 4)
```

Un altre exemple:

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

Per a recuperar l'element d'una llista en una posició determinada s'utilitza el comando `nth$`:

```
(nth$ <integer-expression> <multifield-expression>)
```

Exemples:

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

Per a esborrar un element d'una llista s'utilitza el comando `delete-member$`:

```
(delete-member$ <multifield-expression> <expression>+)
```

Exemples:

```
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>
```

Per a reemplaçar un element d'una llista s'utilitza el comando `replace-member$`

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

Exemples:

```
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>
```

Per a inserir un element en una llista s'utilitza el comando `insert$`:

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

Exemples

```
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>
```

Per a trobar el primer element d'una llista s'utilitza el comando `first$`:

```
(first$ <multifield-expression>)
```

Exemples

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

Per a trobar la resta d'elements d'una llista s'utilitza el comando `rest$`:
`(rest$ <multifield-expression>)`

Exemples

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

Comando `read`

Per a llegir dades d'un fitxer: `(read [<logical-name>])`

Exemples:

```
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`

Per a llegir una línia d'un fitxer: `(readline [<logical-name>])`

Exemples:

```
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>
```