Artificial Intelligence: laboratory
CLIPS

Fall 2008

professor: Luigi Ceccaroni

El sistema CLIPS

- CLIPS es un entorno para desarrollar sistemas expertos.
- Define un lenguaje que permite la representación de:
  - conocimiento declarativo (frames)
  - conocimiento procedimental (reglas de producción)
- Su base es un motor de inferencias con razonamiento hacia adelante.

El lenguaje de CLIPS

- El lenguaje de CLIPS deriva su sintaxis del lenguaje LISP.
- Se trata de un lenguaje parentizado con notación prefija.
- Los tipos de datos predefinidos que interesan son:
  - real
  - entero
  - cadena
  - símbolo
  - apuntador a hechos
  - nombre de instancia
  - apuntador a instancia
- Los tipos habituales poseen los operadores comunes.
- El lenguaje de CLIPS aúna tres paradigmas de programación:
  - lenguaje de reglas
  - lenguaje funcional
  - lenguaje orientado a objetos

Hechos

- Los dos elementos que permiten representar problemas utilizando reglas de producción son los hechos y las reglas.
- Los hechos en CLIPS pueden ser de dos tipos:
  - *ordered facts*
  - *deftemplate facts*
- Los ***ordered facts*** tienen formato libre, por lo tanto no tienen una estructura predefinida, siguen el esquema general:

(*relación* p_1 p_2 ... p_n)

- *relación* ha de ser un símbolo; el resto de parámetros puede ser de cualquier tipo, por ejemplo:

(padre Juan Pedro)

(num-hijos Juan 2)

Hechos

- Los ***deftemplate facts*** tienen una estructura predefinida; pueden asimilarse a representaciones al estilo de los marcos.
- En estos hechos se define una serie de campos (*slots*) que enmarcan su estructura. Cada campo puede tener una serie de restricciones como:
  - tipo
  - cardinalidad
  - un valor por defecto que puede ser:
    - una constante
    - un función para calcularlo
- Sintaxis:

```
(deftemplate nombre-template "comentario“
    (slot nombre-slot)
    (multislot nombre-slot))
```

- Por ejemplo:

```
(deftemplate persona “character in a play”
    (slot nombre (type STRING))
    (slot edad (type INTEGER) (default 0)))
```

Hechos

- La creación de hechos se realiza mediante la sentencia *assert* (uno solo) o *deffacts* (un conjunto), por ejemplo:

```
(assert (padre Pepe Juan))
(assert (persona (nombre "pedro") (edad 25)))
(deffacts mis-hechos
   (casa roja)
   (pelota verde)
   (persona (nombre "luis”) (edad 33)))
```

Hechos

- (facts) permite saber que hechos hay definidos.
- (clear) borra todos los hechos definidos.
- (retract <índice-hecho>) elimina el hecho identificado por el índice dado.
- (get-deftemplate-list) retorna la lista de *deftemplates* definidos.

Reglas

- Las reglas en CLIPS están formadas por:
  - Una parte izquierda (LHS) que define las condiciones a cumplir
  - Una parte derecha (RHS) que define las acciones a realizar
- Sintaxis:

```
(defrule nombre-regla "comentario"
   (condición-1) (condición-2) ...
   =>
   (acción-1) (acción-2) ...)
```

Variables

- Para poder establecer patrones en las condiciones de las reglas hacen falta variables
- Las variables en CLIPS se denotan poniendo un interrogante delante del nombre (?variable)
- Existen variables anónimas (no importa su valor) para un valor ? o para múltiples valores $?
- Durante la ejecución de las reglas se buscan los valores que instancien las variables y permitan cumplir las condiciones
- Las variables de las reglas son locales, si queremos definir variables globales debemos usar la construcción *defglobal* (las variables globales se denotan ?*variable*)

Notation

- symbols, characters, keywords
  - entered exactly as shown:
  - (example)
- square brackets [...]
  - contents are optional:
  - (example [test])
- pointed brackets (less than / greater than signs) < ... >
  - replace contents by an instance of that type
  - (example <char>)
- star *
  - replace with zero or more instances of the type
  - <char>*
- plus +
  - replace with one or more instances of the type
  - <char>+ (is equivalent to <char> <char>*)
- vertical bar |
  - choice among a set of items:
  - true | false

Rules: left-hand side

- In the LHS of a rule, there can be different types of conditions:
  - Patterns including variables and wildcards, which are directly instantiated with facts in the fact base
  - Patterns together with *not*, *and*, *or*, *exist* and *forall* expressions
  - Tests on expressions including assigned variables
- Restrictions over variables can be combined through logical connectives: ~ (not), & (and), | (or)

...

Object orientation in CLIPS

- It is an extension of *deftemplate facts*, which allows using something **similar to ontology classes** for representation.
- Classes can be defined with **slots** and **methods**.
- Classes corresponding to predefined **CLIPS types** are also predefined and hierarchically organized.

Object orientation in CLIPS

- The sentence to define a class is *defclass*, which specifies:
  - The **name/label** of the class
  - A list of **superclasses** (from which slots and methods are inherited) (at least one superclass)
  - If it is abstract or concrete (Instances can be defined.)
  - If its instances can **match patterns** of rule's LHS
  - The slots

Object orientation in CLIPS

- Examples:

```
(defclass living-being
  (is-a living thing)
  (role abstract)
  (pattern-match non-reactive)
  (slot breathe (default yes)))

(defclass person
  (is-a living-being)
  (role concrete)
  (pattern-match reactive)
  (slot nickname))
```

Object orientation in CLIPS - Slots

- Slot's definition includes a set of facets, such as:
  - (type …)
  - (cardinality …)
  - (default …)
  - (access *read-write* | *read-only* | *initialize-only*)
  - (visibility *public* | *private*)
  - (create-accessor *?NONE* | *read* | *write* | *read-write*)

Object orientation in CLIPS - Instances

- To create an instance the sentence *make-instance* is used.
- When an instance is created, a value has to be given to all its slots:

  (make-instance Juan of person (nickname “Juan"))

- Sets of instances can be created with the sentence *definstances*:

  (definstances persons

  (Juan of person (nickname “Juan"))

  (Maria of person (nickname “Maria"))

)

Object orientation in CLIPS - Messages

- All interaction among objects is via **messages**.
- Message's task is processed and carried out by **message *handlers***.
- Handlers are defined with the sentence *defmessage-handler*, with the same syntax as functions

  (defmessage-handler <class> name/label <type> (<param>) <expr>*)*

- We are concerned with ***primary*** *message handlers* only.

Object orientation in CLIPS - Messages

- All classes have a set of handlers defined by default, such as: *init, delete, print*.
- Defining *(create-accessor read-write)* in a slot automatically creates two handlers: **get-*slot_name*** and **set-*slot_name***, to access and modify the value of the slot.
- The slots of the object for which a handler is being defined are accessed via the variable *?self*, placing ":" before the name of the slot:

  (defmessage-handler person write-nickname ()
  (printout t "Name:" ?self:nickname crlf))

Object orientation in CLIPS - Messages

- Messages are sent to instances via the sentence *send*, with the instance's name in square brackets:

  (send [Juan] write-nickname)

  (send [Juan] set-nickname "Pedro")

- Subclasses can execute handlers of superclasses, using the sentence *call-next-handler*.
- Each message has always to include at least a *primary handler*.

Object orientation in CLIPS – Instances and rules

- To use instances in the RHS of a rule, in the LHS the sentence ***object*** is used, which returns a pointer to the underlying CLIPS C object:

  (defrule rule-persons

  …(object (is-a person) (nickname ?x))

  => *...)*

- The modification of an instance’s slot allows to re-instantiate a rule with that instance.

El lenguaje de reglas de CLIPS - ejemplos

- Personas mayores de 18 años: `(persona (edad ?x&:(> ?x 18)))`
- Personas de nombre juan o pedro: `(persona (nombre juan|pedro))`
- Personas con nombres diferentes: `(persona (nombre ?x)) (persona (nombre ?y&~?x))`
- Nadie se llama pedro: `(not (persona (nombre pedro)))`
- Todo el mundo es mayor de edad: `(forall (persona (nombre ?n) (edad ?x)) (test (> ?x 18)))`

El lenguaje de reglas de CLIPS

- Podemos obtener la dirección del hecho que instancia un patrón mediante el operador <-, por ejemplo:

```
(defrule mi-regla
         ?x <- (persona (nombre juan))
        =>
         (retract ?x)
)
```

- En la parte derecha de las reglas podemos poner cualquier sentencia válida en clips (ver manual)

El lenguaje de reglas de CLIPS - módulos

- Las reglas de CLIPS se puede agrupar en módulos para poder organizarlas
- La ventaja principal es el poder estructurar el conocimiento y poder focalizar la ejecución de las reglas según su objetivo
- La definición de un módulo se realiza mediante la construcción `(defmodule <nombre> "comentario" <export-import>)`
- Nada de lo definido en un módulo es visible salvo que lo exportemos
- Para utilizar construcciones de otro módulo tambien tenemos que importarlas explícitamente
- Existe un módulo por defecto llamado MAIN al que pertenece todo lo no definido en otro módulo

El lenguaje de reglas de CLIPS - módulos

- Podemos definir construcciones pertenecientes a un módulo poniendo como prefijo de su nombre el nombre del módulo seguido de dobles dos puntos `::`, por ejemplo:

```
(deftemplate A::cubo (slot tamanyo))
```

- La exportación de construcciones de un módulo se realiza incluyendo la sentencia `export` en su definición. Podemos exportar cualquier cosa que definamos, por ejemplo:

```
(defmodule A (export deftemplate cubo))
(defmodule A (export deftemplate ?ALL))
```

- La importación de construcciones a un módulo se realiza incluyendo la sentencia `import` en su definición. Podemos importar cualquier cosa visible que este definida en otro módulo, por ejemplo:

```
(defmodule B (import A deftemplate cubo))
```

El lenguaje de reglas de CLIPS - foco

- Podemos restringir qué módulos se usan para la ejecución de reglas mediante la sentencia (`focus <modulo>*`)

- Esta sentencia se puede incluir en la parte derecha de una regla para poder cambiar explícitamente de módulo

- Se puede hacer que la ejecución se focalice en el módulo de la última regla ejecutada declarando la propiedad `auto-focus` en una regla, por ejemplo:

```
(defrule  JUAN::mi-regla
          (declare (auto-focus TRUE))
          (persona (nombre juan))
     => ...
```

Estrategias de resolución de conflicto

El intérprete de reglas tiene definidas unas estrategias de resolución de conflicto

- Profundidad, las nuevas activaciones pasan al principio
- Anchura, las nuevas activaciones pasan al final
- Simplicidad, ante la misma posibilidad de activar, se prefiere las menos específicas (especificidad medida respecto a la complejidad de las condiciones)
- Complejidad, tienen preferencia las reglas más específicas

Estrategias de resolución de conflicto

- Estrategia LEX, recencia de los hechos instanciados, tomando los hechos instanciados ordenadamente en cada regla y siguiendo orden lexicogáfico de recencia

- Estrategia MEA, Se ordenan por recencia respecto al hecho que instancia la primera condición, en caso de empate se sigue la estrategia LEX

- Aleatoria, se disparan las reglas en orden aleatorio

El lenguaje de programación de CLIPS

- CLIPS incluye un lenguaje de programación pseudo-funcional
- Éste permite definir nuevas funciones o programar las acciones a realizar en la parte derecha de las reglas
- Toda sentencia o estructura de control es una función que recibe unos parámetros y retorna un resultado (paradigma funcional)

El lenguaje de programación de CLIPS - Sentencias

Estas son las sentencias y estructuras de control mas utilizadas:

- (bind <var> <valor>): Asignación a una variable, retorna el valor asignado
- (if <exp> then <accion>* [else <accion*>]): Sentencia alternativa, retorna el valor de la última acción evaluada
- (while <exp> do <accion>*): Bucle condicional, retorna falso, excepto si hay una sentencia de retorno que rompa el bucle
- (loop-for-count (<var> <val-i> <val-f>) do <accion>*): Bucle sobre un rango de valores, retorna falso, excepto si hay una sentencia de retorno

El lenguaje de programación de CLIPS - Sentencias

- (progn <accion>*): Ejecuta un conjunto de sentencias secuencialmente, retorna el valor de la última
- (return <expr>): Rompe la ejecución de la estructura de control que la contiene retornando el valor de la expresión
- (break): Rompe la ejecución de una estructura de control
- (switch <expr> (case (<comp>) then <accion>*)* [(default <accion>*)]): Estructura alternativa caso, cada case hace una comparación con el valor evaluado. Retorna la última expresión evaluada o falso si ninguna sentencia case se cumple

El lenguaje de programación de CLIPS - definir funciones

- La construcción `deffunction` permite definir nuevas funciones

```
(deffunction <nombre> "Comentario"
    (<?parametro>* [<$?parametro-wilcard>])
     <accion>*)
```

- La lista de parámetros puede ser variable, el parametro wilcard incluye en una lista el resto de parámetros

- La función retorna la última expresión evaluada

Orientación a objetos en CLIPS

- CLIPS define también una extensión orientada a objetos que complementa la capacidad de representar la estructura del conocimiento
- Se puede considerar como una extensión del constructor `deftemplate` que pretende completar la posibilidad de usar *frames* como herramienta de representación
- Podemos definir clases como en los lenguajes orientados a objetos con slots y métodos
- CLIPS tiene definido un conjunto inicial de clases que organizan los tipos predefinidos de CLIPS estableciendo una jerarquía entre ellos

Orientación a objetos en CLIPS

- La sentencia que permite definir una clase es `defclass`
- Para definir una clase hay que especificar:
  1. El nombre de la clase
  2. Una lista de sus superclases (heredará de estas sus slots y métodos)
  3. Declaración de si es una clase abstracta o no (permitimos definir instancias)
  4. Si permitimos que instancias de esta clase puedan vincularse a patrones en la LHS de una regla
  5. Definición de los slots de la clase (slot, multi-slot)
- Toda clase debe tener como mínimo una superclase

Orientación a objetos en CLIPS

Por ejemplo:

```
(defclass ser-vivo
          (is-a USER)
          (role abstract)
          (pattern-match non-reactive)
          (slot respira (default si)))

(defclass persona
         (is-a ser-vivo)
         (role concrete)
         (pattern-match reactive)
         (slot nombre))
```

Orientación a objetos en CLIPS - slots

La definición de un slot incluye un conjunto de propiedades, algunas son:

- `(default ?DERIVE|?NONE|<exp>*)`
- `(default-dynamic <expr>*)`
- `(access read-write|read-only|initialize-only)`
- `(propagation inherit|no-inherit)`
- `(visibility public|private)`
- `(create-accessor ?NONE|read|write|read-write)`
- Tambien se puede declarar el tipo, cardinalidad, ...

Orientación a objetos en CLIPS - instancias

- Para crear instancias de una clase se usa la sentencia `make-instance`
- Al crear una instancia debemos dar valor a los slots que posee, por ej:

```
(make-instance juan of persona (nombre "juan"))
```

- Podemos crear conjuntos de instancias con la sentencia `definstances`, por ej:

```
(definstances personas
   (juan of persona (nombre "juan"))
   (maria of persona (nombre "maria"))
)
```

Orientación a objetos en CLIPS - mensajes

- Toda la interacción con los objetos se realiza mediante lo que se denomina mensajes

- Estos mensajes tienen manejadores (*message handlers*) que los procesan y realizan la tarea indicada

- Estos *manejadores* se definen mediante la sentencia `defmessage-handler`, su sintaxis es idéntica a la de las funciones.

```
(defmessage-handler <clase> nombre <tipo-h> (<param>*) <expr>*)
```

- Existen diferentes tipos de manejadores `primary`, `before`, `after`, `around`, nosotros sólo nos preocuparemos de los `primary`

Orientación a objetos en CLIPS - mensajes

- Por defecto toda clase tiene definidos un conjunto de manejadores, por ejemplo: `init`, `delete`, `print`

- Al definir `create-accessor` en un slot estamos creando dos mensajes, `get-nombre_slot`, `set-nombre_slot` para acceder y modificar el slot

- El acceso a los slots de un objeto dentro de un manejador se realiza mediante la variable `?self`, poniendo `:` delante del nombre del slot, por ejemplo:

```
(defmessage-handler persona escribe-nombre ()
   (printout t "Nombre:" ?self:nombre crlf))
```

Orientación a objetos en CLIPS - mensajes

- El envío de los mensajes a una instancia se realiza mediante la sentencia `send`, el nombre de la instancia se pone entre corchetes, por ejemplo:

```
(send [juan] escribe-nombre)
(send [juan] set-nombre "pedro")
```

- Los manejadores se pueden definir en cada clase, por lo tanto las subclases pueden ejecutar los manejadores de sus superclases. Para los de tipo `primary` estos se inician desde la clase más específica, si se quiere ejecutar los de las superclases se ha de usar la sentencia `call-next-handler`

- Debe haber siempre como mínimo un manejador `primary` para cada mensaje

Orientación a objetos en CLIPS - instancias y reglas

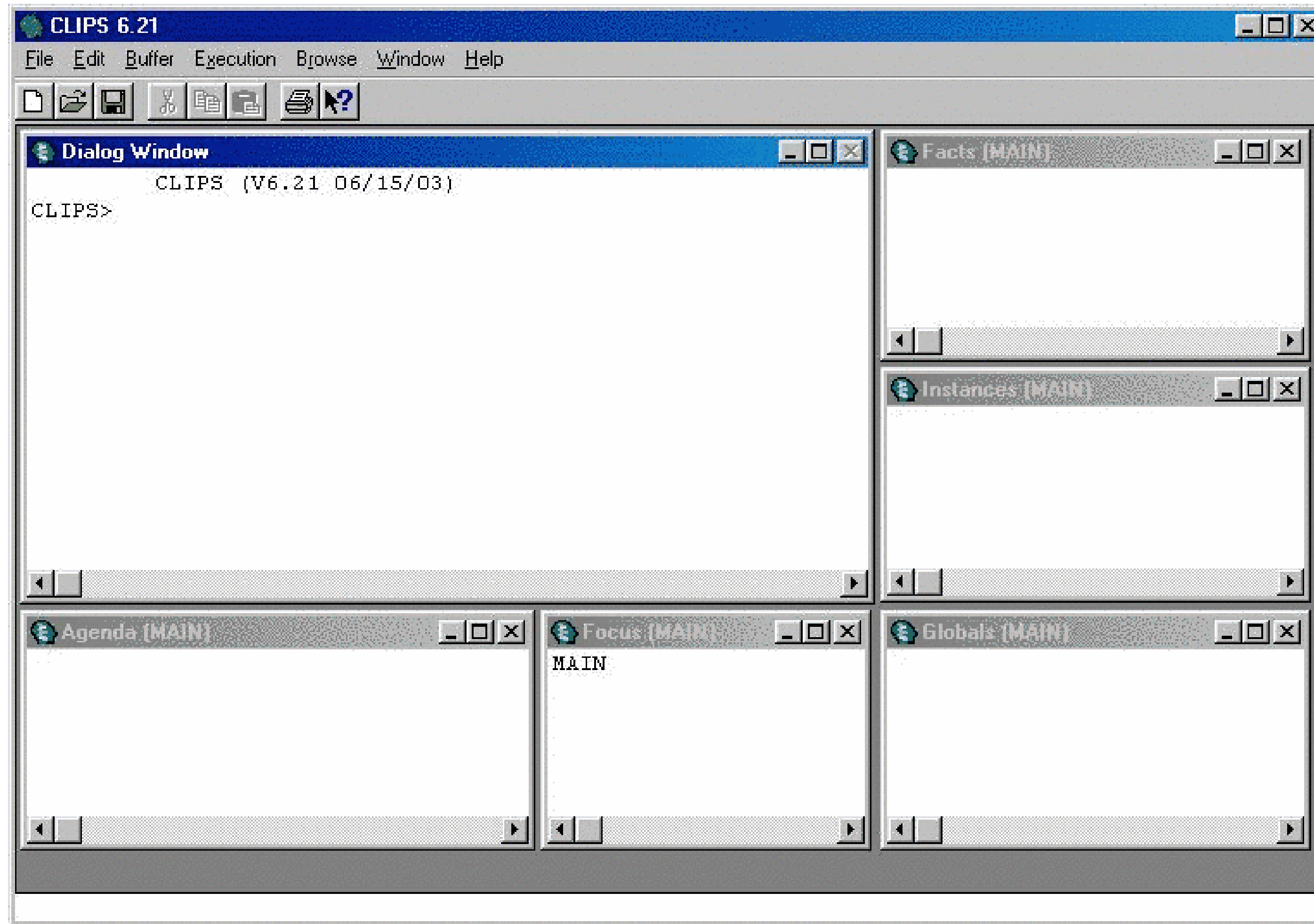
- Para poder usar instancias en la RHS de una regla se utiliza la sentencia `object`, por ejemplo:

```
(defrule regla-personas
      (object (is-a persona) (nombre  ?x))
 =>
  ...
)
```

- La clase se ha de haber declarado como utilizable en la LHS de las reglas

- La modificación de un slot de una instancia vuelve a permitir que se pueda volver a instanciar una regla con ella

CLIPS - La aplicación

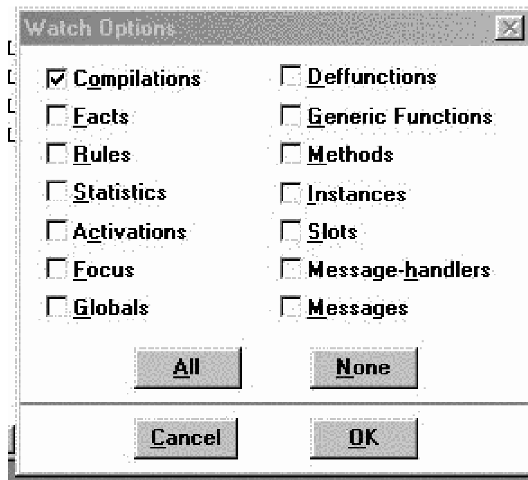


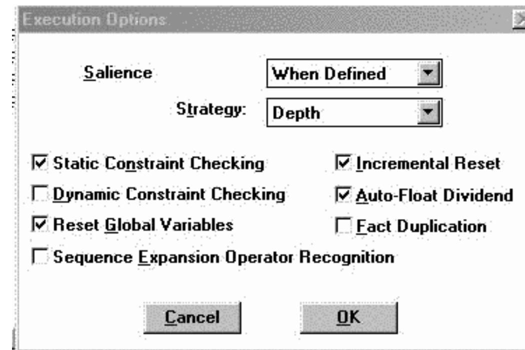
Manejo del intérprete de CLIPS - ventanas

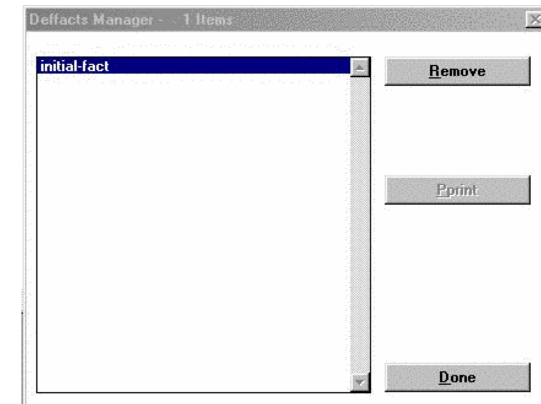
- Ventana del intérprete
- Ventana de hechos (hechos definidos y obtenidos durante la ejecución)
- Ventana de instancias (instancias de las clases definidas)
- Ventana de defglobals (variables globales)
- Ventana del foco (módulos en los que está el foco)
- Ventana de la agenda (activaciones posibles)

Manejo del intérprete de CLIPS

A través de los menus se puede acceder a los comandos mas comunes (ejecución, parada, debugging, visualización de definiciones, configuración)







Manejo del intérprete de CLIPS - comandos

- `(facts)`, lista los hechos que hay en la base de hechos
- `(reset)`, borra todos los hechos que hay en la base de hechos
- `(clear)`, borra todos los hechos y las reglas
- `(rules)`, lista las reglas definidas
- `(load "nombre.clp")`, carga un fichero CLIPS
- `(save "nombre.clp")`, salva todos los deffacts, defrule y deffunction en un fichero

Manejo del intérprete de CLIPS - ejecución

- `(run [<entero>])`, ejecuta el motor de inferencia, un número de pasos o hasta el final, si no se pasa ningún parámetro
- `(agenda [<modulo]>)`, muestra todas las posibles activaciones o sólo las de un módulo
- `(focus <modulo>+)`, pone el foco de ejecución en los módulos indicados
- `(halt)`, para la ejecucion del motoe de inferencias
- `(set-strategy <estrategia>)`, cambia la estrategia de resolución de conflictos

Manejo del intérprete de CLIPS - debugging

- (watch <item>), informa sobre <item> durante la ejecución, donde <item> puede ser entre otros:
  - facts <hechos>*, informa sobre los hechos indicados
  - rules <regla>*, informa sobre las reglas indicadas
  - activations <regla>*, informa sobre la activación de las reglas indicadas
  - deffunctions <funcion>*, informa sobre las funciones indicadas
  - all, informa sobre todo
- (unwatch <item>), elimina el watch correspondiente

Manejo del intérprete de CLIPS - debugging

- `(set-break <regla>)`, introduce un punto de ruptura de ejecucion en la regla

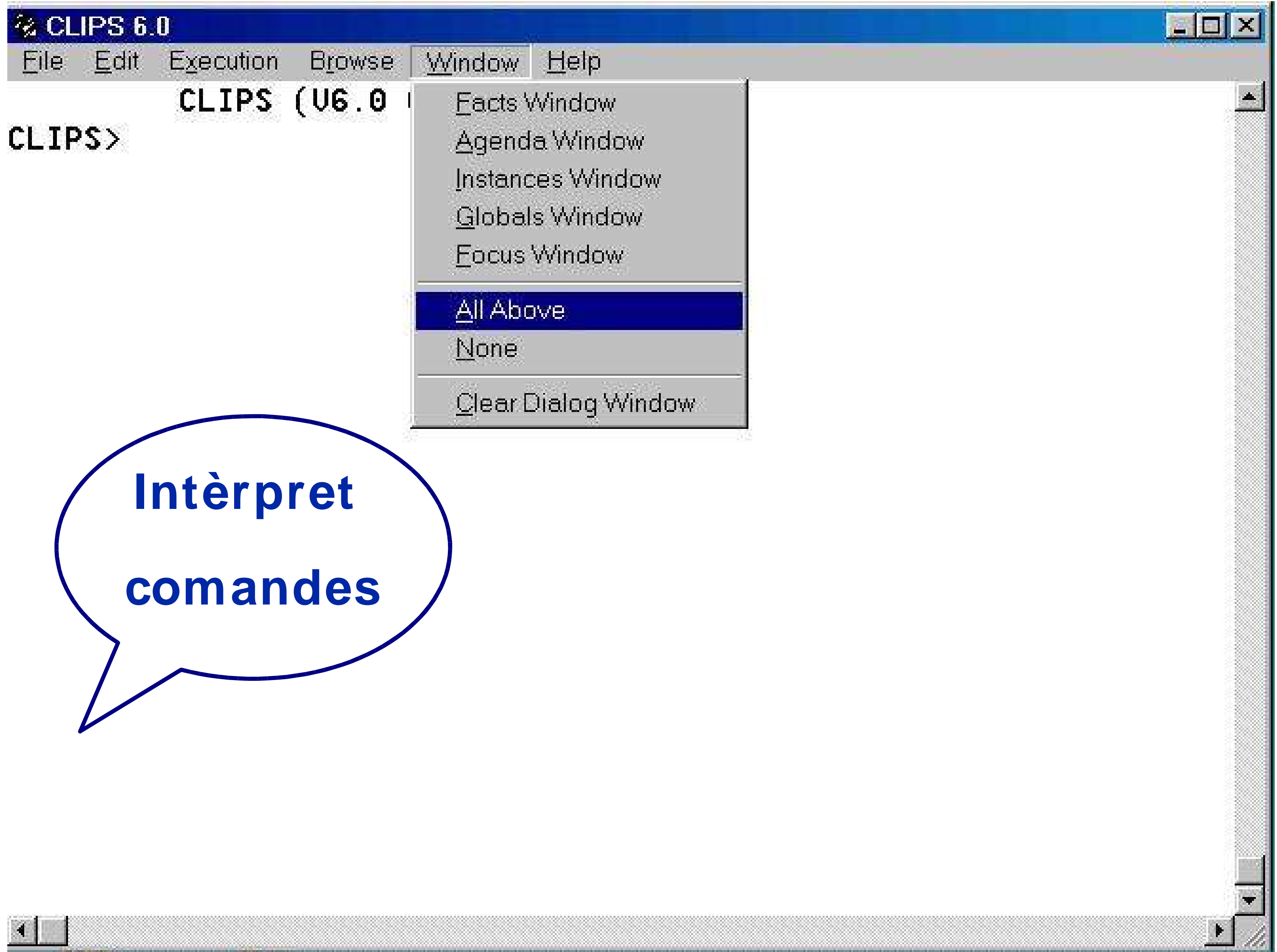
- `(remove-break <regla>)`, elimina un punto de ruptura de ejecucion en la regla

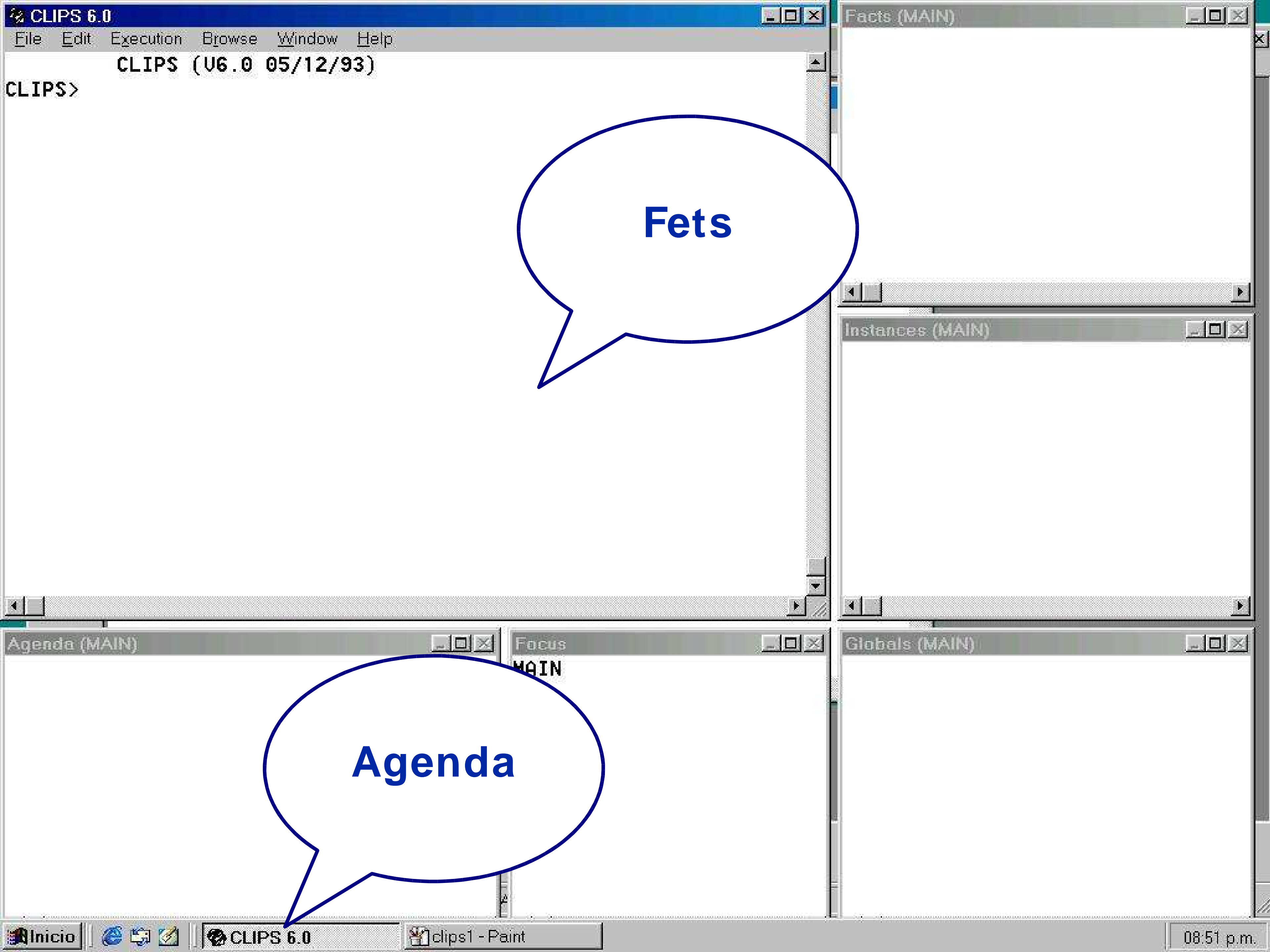
- `(show-breaks)`, muestra los puntos de ruptura definidos

- `(matches <regla>)`, indica las instanciaciones posibles para una regla

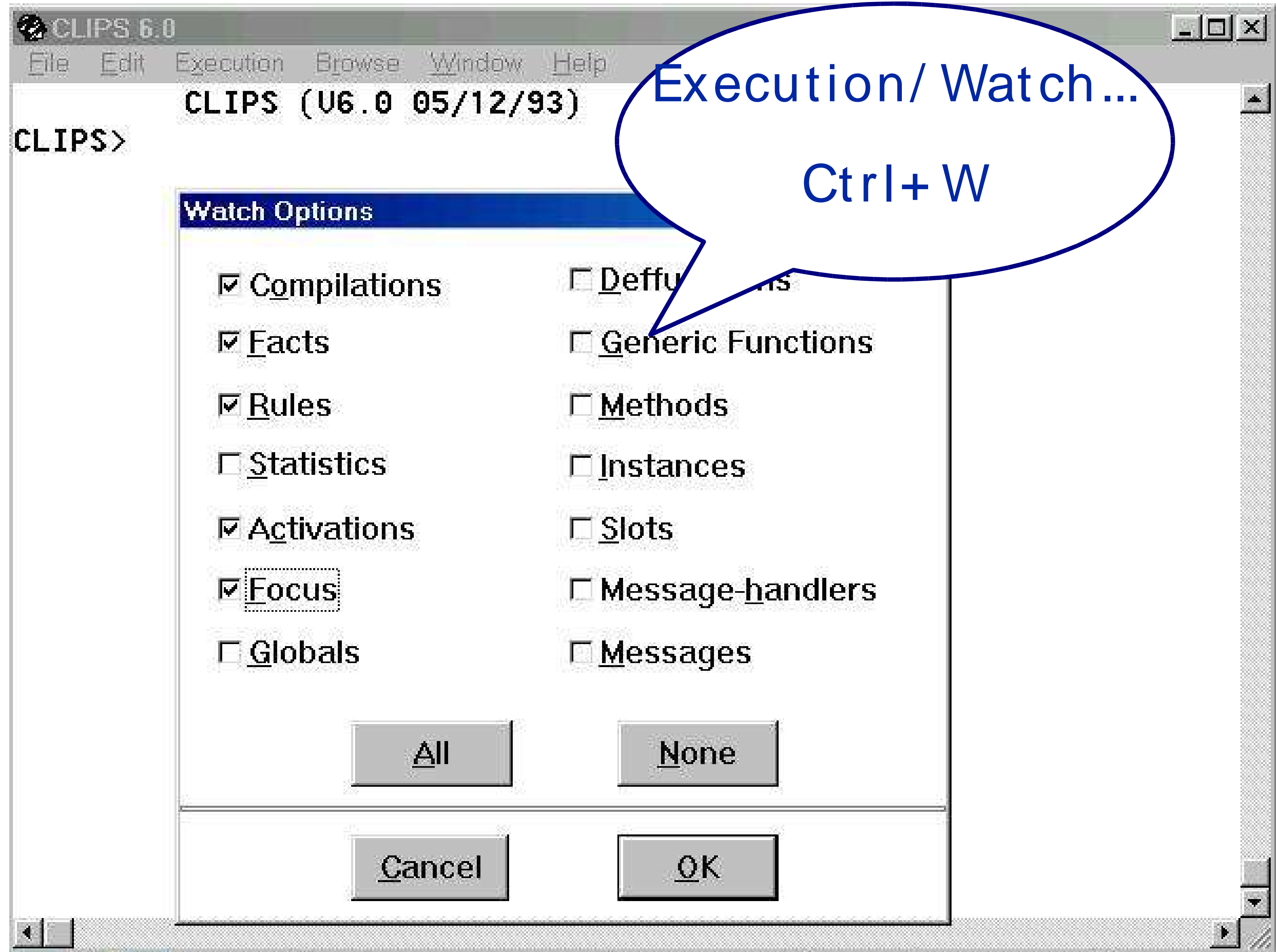
- `(dribble-on <fichero>)` y `(dribble-off)`, redirecciona la información de la traza a un fichero y elimina la redirección

CLIPS: Exemple

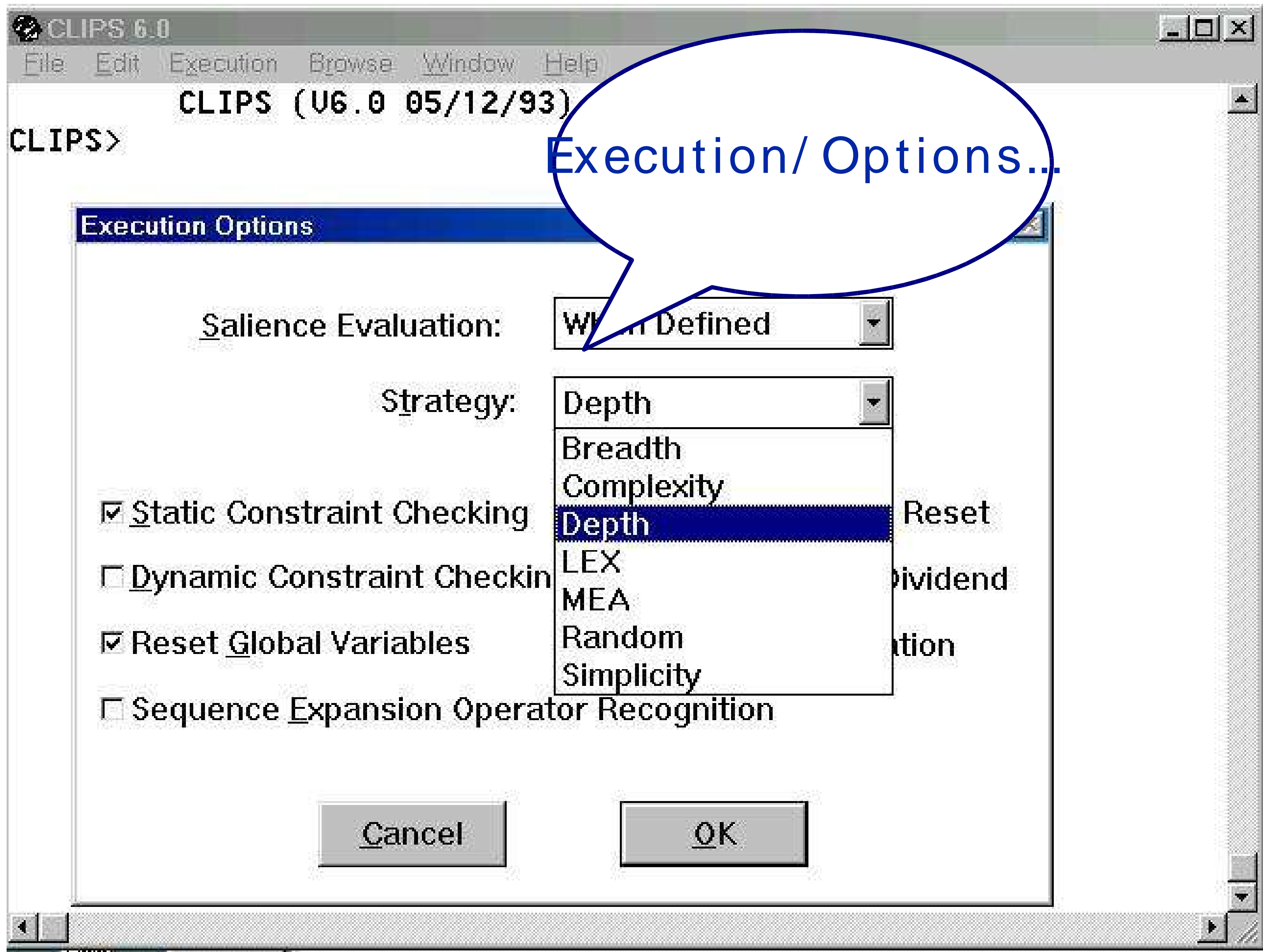



CLIPS 6.0
File Edit Execution Browse Window Help
CLIPS (V6.0 05/12/93)
CLIPS>
Fets
Facts (MAIN)
Instances (MAIN)
Agenda (MAIN)
Focus
MAIN
Agenda
Globals (MAIN)
Inicio
CLIPS 6.0
clips1 - Paint
08:51 p.m.

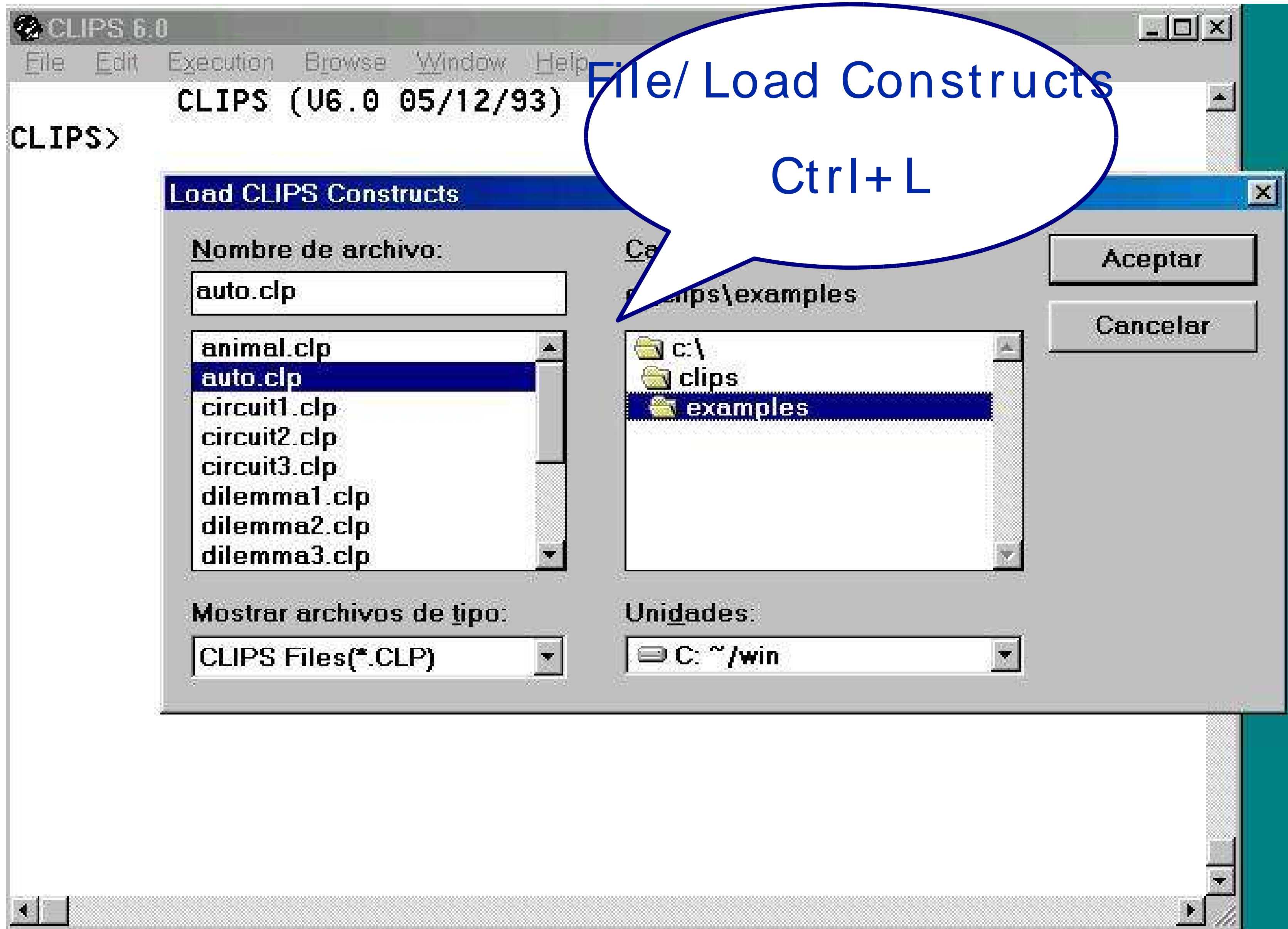
CLIPS: Exemple



CLIPS: Exemple



CLIPS: Exemple



CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
CLIPS> Defining deffunction: ask-qu
Defining deffunction: yes-or-no-p
Defining defrule: normal-engine-sta
Defining defrule: unsatisfactory-en
Defining defrule: determine-engine-
Defining defrule: determine-rotatic
Defining defrule: determine-sluggis
Defining defrule: determine-misfiri
Defining defrule: determine-knockir
Defining defrule: determine-low-out
Defining defrule: determine-gas-level =j+
Defining defrule: determine-battery-state
Defining defrule: determine-point-surface
+j+j
Defining defrule: determine-conductivity-
Defining defrule: no-repairs =j+j
Defining defrule: system-banner =j
Defining defrule: print-rep
TRUE
CLIPS> (reset)
CLIPS>
```

Facts (MAIN)

```
f-0     (initial-fact)
```

Agenda (MAIN)

```
10      system-banner: f-0
0       determine-engine-state: f-0,,
-10     no-repairs: f-0,
```

(reset)

CLIPS: Exemple

File/ Editor...

(Ctrl+E)

File/ Open...

CLIPS 6.0

File Edit Execution Browse Window Help

```
CLIPS> (load "C:/CLIPS/EXAMPLES/AUTO.C
CLIPS> Defining deffunction: ask-ques
Defining deffunction: yes-or-no-p
Defining c
Defining c
Defining c
Defining c
Defining c
Defining c
Defining c
Defining c
Defining c
Defining c
Defining c
+j+j
Defining c
Defining c
Defining c
Defining c
TRUE
CLIPS>
```

Edit File - (AUTO.CLP)

File Edit Buffer Help

```
;;;======================================================
;;;   Automotive Expert System
;;;
;;;     This expert system diagnoses some simple
;;;     problems with a car.
;;;
;;;     CLIPS Version 6.0 Example
;;;
;;;     To execute, merely load, reset and run.
;;;======================================================

;;****************
;;* DEFFUNCTIONS *
;;****************

(deffunction ask-question (?question $?allowed-values)
   (printout t ?question)
```

CLIPS: Exemple

```
;;****************
;;* DEFFUNCTIONS *
;;****************

(deffunction ask-question (?question $?allowed-values)
   (printout t ?question)
   (bind ?answer (read))
   (if (lexemep ?answer)
       then (bind ?answer (lowcase ?answer)))
   (while (not (member ?answer ?allowed-values)) do
      (printout t ?question)
      (bind ?answer (read))
      (if (lexemep ?answer)
          then (bind ?answer (lowcase ?answer))))
   ?answer)

(deffunction yes-or-no-p (?question)
   (bind ?response (ask-question ?question yes no y n))
   (if (or (eq ?response yes) (eq ?response y))
       then TRUE
       else FALSE))
```

CLIPS: Exemple

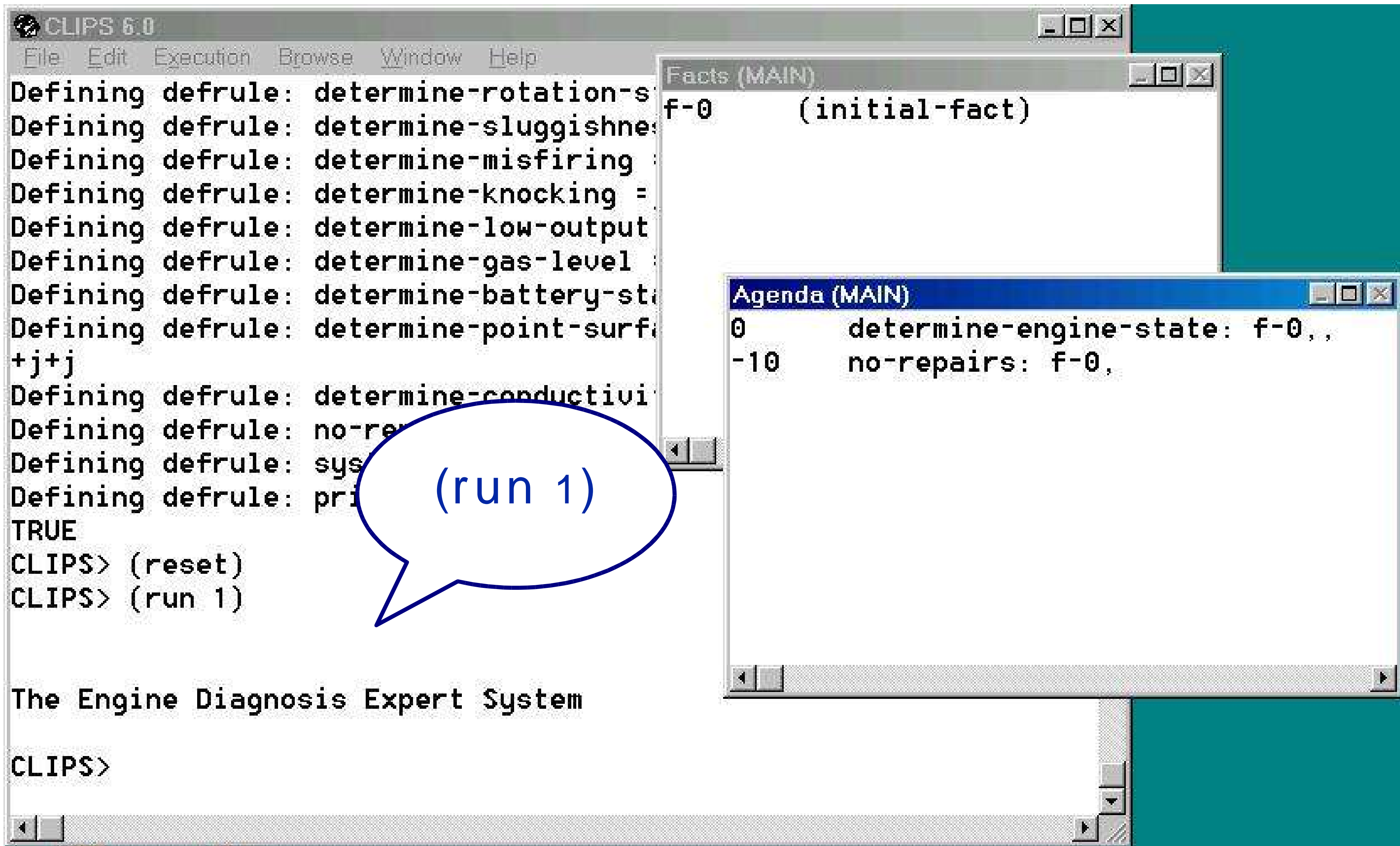
```
(defrule no-repairs ""
  (declare (salience -10))
  (not (repair ?))
  =>
  (assert (repair "Take your car to a mechanic.")))

;;;****************************
;;;* STARTUP AND REPAIR RULES *
;;;****************************

(defrule system-banner ""
  (declare (salience 10))
  =>
  (printout t crlf crlf)
  (printout t "The Engine Diagnosis Expert System")
  (printout t crlf crlf))

(defrule print-repair ""
  (declare (salience 10))
  (repair ?item)
  =>
  (printout t crlf crlf)
  (printout t "Suggested Repair:")
  (printout t crlf crlf)
  (format t " %s%n%n%n" ?item))
```

CLIPS: Exemple



CLIPS: Exemple

```
;;;***************
;;;* QUERY RULES *
;;;***************

(defrule determine-engine-state ""
   (not (working-state engine ?))
   (not (repair ?))
   =>
   (if (yes-or-no-p "Does the engine start (yes/no)? ")
       then
       (if (yes-or-no-p "Does the engine run normally (yes/no)? ")
           then (assert (working-state engine normal))
           else (assert (working-state engine unsatisfactory)))
       else
       (assert (working-state engine does-not-start))))
```

CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
Defining defrule: determine-sluggishne
Defining defrule: determine-misfiring
Defining defrule: determine-knocking =
Defining defrule: determine-low-output
Defining defrule: determine-gas-level
Defining defrule: determine-battery-st
Defining defrule: determine-point-surf
+j+j
Defining defrule: determine-conductivi
Defining defrule: no-repairs =j+j
Defining defrule: system-banner =j
Defining defrule: print-repair +j
TRUE
CLIPS> (reset)
CLIPS> (run 1)


The Engine Diagnosis Expert System

CLIPS> (run 1)
Does the engine start (yes/no)? no
```

Facts (MAIN)

```
f-0     (initial-fact)
```

Agenda (MAIN)

```
-10    no-repairs: f-0,      : f-0,,
```

CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
Defining defrule: determine-mis
Defining defrule: determine-kno
Defining defrule: determine-low
Defining defrule: determine-gas
Defining defrule: determine-bat
Defining defrule: determine-poi
+j+j
Defining defrule: determine-con
Defining defrule: no-repairs =j
Defining defrule: system-banner
Defining defrule: print-repair
TRUE
CLIPS> (reset)
CLIPS> (run 1)


The Engine Diagnosis Expert System

CLIPS> (run 1)
Does the engine start (yes/no)? no
CLIPS>
```

Facts (MAIN)

```
f-0      (initial-fact)
f-1      (working-state engine does-not-start)
```

Agenda (MAIN)

```
0        determine-rotation-state: f-1,,
-10      no-repairs: f-0,
```

CLIPS: Exemple

```
(defrule determine-rotation-state ""
   (working-state engine does-not-start)
   (not (rotation-state engine ?))
   (not (repair ?))
   =>
   (if (yes-or-no-p "Does the engine rotate (yes/no)? ")
       then
       (assert (rotation-state engine rotates))
       (assert (spark-state engine irregular-spark))
       else
       (assert (rotation-state engine does-not-rotate))
       (assert (spark-state engine does-not-spark))))
```

CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
Defining defrule: determine-low
Defining defrule: determine-gas
Defining defrule: determine-ba
Defining defrule: determine-po
+j+j
Defining defrule: determine-con
Defining defrule: no-repairs =j
Defining defrule: system-banner
Defining defrule: print-repair
TRUE
CLIPS> (reset)
CLIPS> (run 1)


The Engine Diagnosis Expert System

CLIPS> (run 1)
Does the engine start (yes/no)? no
CLIPS> (run 1)
Does the engine rotate (yes/no)? yes
CLIPS>
```

Facts (MAIN)

```
f-0     (initial-fact)
f-1     (working-state engine does-not-start)
f-2     (rotation-state engine rotates)
f-3     (spark-state engine irregular-spark)
```

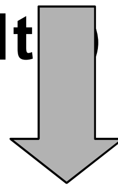
Agenda (MAIN)

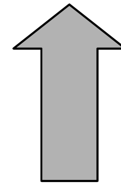
```
0      determine-point-surface-state: f-
0      determine-gas-level: f-1,f-2,
-10    no-repairs: f-0,
```

CLIPS: Exemple

```
Agenda (MAIN)
0       determine-point-surface-state: f-1,f-3,
0       determine-gas-level: f-1,f-2,
-10     no-repairs: f-0,
```

(set- strategy **breadt**)





(set- strategy **depth**)

```
Agenda (MAIN)
0       determine-gas-level: f-1,f-2,
0       determine-point-surface-state: f-1,f-3,
-10     no-repairs: f-0,
```

CLIPS: Exemple

```
(defrule determine-point-surface-state ""
   (or (and (working-state engine does-not-start)
            (spark-state engine irregular-spark))
       (symptom engine low-output))
   (not (repair ?))
   =>
   (bind ?response
      (ask-question "What is the surface state of the points (normal/burned/cor
                    normal burned contaminated))
   (if (eq ?response burned)
       then
      (assert (repair "Replace the points."))
       else (if (eq ?response contaminated)
                then (assert (repair "Clean the points.")))))
```

CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
Defining defrule: determine-bat
Defining defrule: determine-poi
+j+j
Defining defrule: determine-con
Defining defrule: no-repairs =j
Defining defrule: system-banner
Defining defrule: print-repair
TRUE
CLIPS> (reset)
CLIPS> (run 1)


The Engine Diagnosis Expert System

CLIPS> (run 1)
Does the engine start (yes/no)? no
CLIPS> (run 1)
Does the engine rotate (yes/no)? yes
CLIPS> (run 1)
What is the surface state of the points (normal/burned/contaminated)? normal
CLIPS>
```

Facts (MAIN)

```
f-0     (initial-fact)
f-1     (working-state engine does-not-start)
f-2     (rotation-state engine rotates)
f-3     (spark-state engine irregular-spark)
```

Agenda (MAIN)

```
0       determine-gas-level: f-1,f-2,
-10     no-repairs: f-0,
```

CLIPS: Exemple

```
(defrule determine-gas-level ""
   (working-state engine does-not-start)
   (rotation-state engine rotates)
   (not (repair ?))
   =>
   (if (not (yes-or-no-p "Does the tank have any gas in it (yes/no)? "))
       then
       (assert (repair "Add gas."))))
```

CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
+j+j
Defining defrule: determine-con
Defining defrule: no-repairs =j
Defining defrule: system-banner
Defining defrule: print-repair
TRUE
CLIPS> (reset)
CLIPS> (run 1)


The Engine Diagnosis Expert Sys

CLIPS> (run 1)
Does the engine start (yes/no)? no
CLIPS> (run 1)
Does the engine rotate (yes/no)? yes
CLIPS> (run 1)
What is the surface state of the points (normal/burned/contaminated)? normal
CLIPS> (run 1)
Does the tank have any gas in it (yes/no)? no
CLIPS>
```

Facts (MAIN)

```
f-0     (initial-fact)
f-1     (working-state engine does-not-start)
f-2     (rotation-state engine rotates)
f-3     (spark-state engine irregular-spark)
f-4     (repair "Add gas.")
```

Agenda (MAIN)

```
10      print-repair: f-4
```

CLIPS: Exemple

```
CLIPS 6.0
File  Edit  Execution  Browse  Window  Help

The Engine Diagnosis Expert Sys

CLIPS> (run 1)
Does the engine start (yes/no)?
CLIPS> (run 1)
Does the engine rotate (yes/no)
CLIPS> (run 1)
What is the surface state of th
CLIPS> (run 1)
Does the tank have any gas in i
CLIPS> (run 1)


Suggested Repair:

 Add gas.


CLIPS>
```

```
Facts (MAIN)
f-0     (initial-fact)
f-1     (working-state engine does-not-start)
f-2     (rotation-state engine rotates)
f-3     (spark-state engine irregular-spark)
f-4     (repair "Add gas.")
```

```
Agenda (MAIN)
```

CLIPS: Exemple

```
(defrule no-repairs ""
  (declare (salience -10))
  (not (repair ?))
  =>
  (assert (repair "Take your car to a mechanic.")))

;;;****************************
;;;* STARTUP AND REPAIR RULES *
;;;****************************

(defrule system-banner ""
  (declare (salience 10))
  =>
  (printout t crlf crlf)
  (printout t "The Engine Diagnosis Expert System")
  (printout t crlf crlf))

(defrule print-repair ""
  (declare (salience 10))
  (repair ?item)
  =>
  (printout t crlf crlf)
  (printout t "Suggested Repair:")
  (printout t crlf crlf)
  (format t " %s%n%n%n" ?item))
```

CLIPS: Exemple

```
CLIPS 6.0
File  Edit  Execution  Browse  Window  Help

Suggested Repair:

 Add gas.


CLIPS> (reset)
CLIPS> (run   1)


The Engine Diagnosis Expert Sys

CLIPS> (run   1)
Does the engine start (yes/no)? no
CLIPS> (run   1)
Does the engine rotate (yes/no)? yes
CLIPS> (run   1)
What is the surface state of the points (normal/burned/contaminated)? normal
CLIPS> (run   1)
Does the tank have any gas in it (yes/no)? yes
CLIPS>
```

```
Facts (MAIN)
f-0      (initial-fact)
f-1      (working-state engine does-not-start)
f-2      (rotation-state engine rotates)
f-3      (spark-state engine irregular-spark)
```

```
Agenda (MAIN)
-10     no-repairs: f-0,
```

CLIPS: Exemple

CLIPS 6.0

File Edit Execution Browse Window Help

```
The Engine Diagnosis Expert Sys

CLIPS> (run   1)
Does the engine start (yes/no)?
CLIPS> (run   1)
Does the engine rotate (yes/no)
CLIPS> (run   1)
What is the surface state of th
CLIPS> (run   1)
Does the tank have any gas in i
CLIPS> (run   1)
CLIPS> (run   1)


Suggested Repair:

 Take your car to a mechanic.


CLIPS>
```

Facts (MAIN)

```
f-0      (initial-fact)
f-1      (working-state engine does-not-start)
f-2      (rotation-state engine rotates)
f-3      (spark-state engine irregular-spark)
f-4      (repair "Take your car to a mechanic.")
```

Agenda (MAIN)