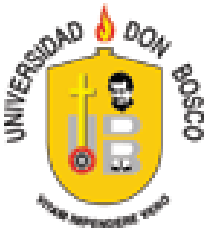
UNIVERSIDAD DON BOSCO

FACULTAD DE INGENIERÍA

ESCUELA DE COMPUTACIÓN



CLISP

INVESTIGACIÓN REALIZADA POR

EVA MARÍA PÉREZ MAJANO **CARNÉT PM-981124**

ERNESTO ALEXANDER PALACIOS MORÁN **CARNTÉT PM-970287**

PARA LA ASIGNATURA DE: SISTEMAS EXPERTOS

CATEDRÁTICO ENCARGADO

ING. CRUZ GALDAMEZ RIVERA

CICLO 02 AÑO 2004

HISTORIA DE CLISP

Los orígenes del "C Language Integrated Production System" (CLISP) datan de 1984 en el centro del espacio de NASA.

En este tiempo, la sección de inteligencia artificial se había interiorizado sobre el uso de las docenas de prototipos, usando el hardware y software disponible para la época.
Sin embargo, a pesar de las demostraciones extensas del potencial de los sistemas expertos, pocos de estos usos fueron puestos en forma regular. Esta falta de tecnología de los sistemas expertos dentro de los apremios operacionales de la NASA, se podía remontar en gran parte al uso del lisp como lenguaje para casi todas las herramientas del software en aquella época.

En detalle, tres problemas obstaculizaron el uso de las herramientas basadas en lisp dentro de la NASA:

a) **La disponibilidad baja del lisp** en una variedad amplia de computadoras convencionales
b) **Alto costo** de herramientas avanzadas y de hardware del lisp,
c) **Integración pobre del lisp** con otros lenguajes.

Llegó a ser evidente que la sección de la inteligencia artificial tendría que desarrollar su propia herramienta basada en los sistemas expertos. Y así fue: El prototipo de CLISP fue desarrollado en dos meses en el año de 1985.

Dónde la característica particular era: Crear una herramienta completamente compatible con los sistemas expertos que se utilizan en inteligencia artificial.

Así, la sintaxis de CLISP fue hecho para asemejarse muy de cerca a la sintaxis de un subconjunto de la herramienta del sistema experto del ART desarrollada por Inference Corporation.
Aunque estuvieron modelados originalmente de ART, CLISP fue desarrollado enteramente sin ayuda de la inferencia o del acceso al código de fuente del ART.

El intento original para CLISP era ganar la penetración y el conocimiento útil sobre la construcción de las herramientas y poner la base para la construcción de una herramienta de reemplazo para las herramientas comerciales que eran utilizadas actualmente.

VERSIONES DE CLISP

La versión 1,0 demostró la viabilidad del concepto del proyecto.

Después del desarrollo adicional, llegó a ser evidente que CLISP sería una herramienta de bajo costo ideal para los propósitos del entrenamiento.

Otro año del desarrollo y del uso interno mejoraban su portabilidad, funcionamiento, funcionalidad, y documentación de soporte.

La versión 3,0 de CLISP fue puesta a disposición fuera de la NASA en el verano de 1986.

Otros realces transformaron CLISP de una herramienta de entrenamiento en una herramienta útil para el desarrollo.

Versiones 4,0 y 4,1 de CLISP, de manual de referencia lanzado de CLISP respectivamente en el verano y a finales de 1987.

La versión 4,2 de CLISP, lanzada en el verano de 1988, era una reescritura completa de CLISP para la modularidad del código.

También fueron incluidos con este lanzamiento un manual de la arquitectura que proporcionaba una descripción detallada de la arquitectura del software de CLISP y de un programa utilitario para ayudar en la verificación y la validación de programas basados en las reglas.

La versión 4,3 de CLISP, lanzada en el verano de 1989, agregó más funcionalidad. Originalmente, la metodología primaria de la representación en CLISP era un lenguaje basado en el algoritmo de Rete.

La versión 5,0 de CLISP, lanzada en 1991, introdujo dos nuevos paradigmas de programación:

a) programación procedural (según lo encontrado en lenguajes tales como C y Ada)
b) programación orientada a objetos.

La versión 5,1 de CLISP, lanzada a fines de 1991, era sobre todo una mejora del software requerida para apoyar las interfaces desarrolladas y/o realzadas de Windows, de MSDOS, y de Macintosh.

Versión 6,0 de CLISP, lanzada en 1993, ayuda para el desarrollo de programas modulares e integración entre las capacidades de programación orientadas a objetos y basadas en las reglas de CLISP.

Versión 6,1 de CLISP, lanzada en 1998, mejora para compiladores NO-ANSI C y para los compiladores de C++.

CLISP ahora se mantiene independientemente de la NASA como software de "public domain" (A lo Mito), debido a su portabilidad, extensibilidad, capacidades, y costo, CLISP ha recibido la aceptación extensa a través del gobierno, de la industria.

En su versión actual es una implementación muy completa de Lisp. Se ejecuta en ordenadores personales (DOS, OS/2, Windows NT, Windows 95, Amiga 500-4000, Acorn RISC PC) así como bajo UNIX (Linux, SVR4, Sun4, DEC Alpha OSF, HP-UX, NeXTstep, SGI, AIX, Sun3 y otros), requiriendo sólo 2 MB de RAM.

CLISP incluye una página de ayuda, al estilo de las páginas de manual de UNIX; y otra en la que se describe la implementación.

El desarrollo de CLISP ha ayudado a mejorar la capacidad de entregar tecnología a sistema expertos a través de los sectores públicos y privados para una amplia gama de usos y de ambientes.

En esto el prototipo para la concepción del lenguaje ha sido la estructura de las funciones matemáticas.
Todos sabemos cómo resolver una expresión del tipo (8 * ((17 + 3) / 4)).

Primero hallaríamos el resultado de 17 + 3, que entonces dividiríamos entre 4, para el resultado multiplicarlo por 8. Es decir, que iríamos resolviendo los paréntesis más interiores y pasando los resultados a las operaciones descritas en los paréntesis que los contienen.

(* 8 (/ (+ 3 17) 4)) sería la función LISP equivalente.

*, / y + son nombres de funciones LISP. Los números en (+ 3 17) son los argumentos que se pasan a la función '+'. Pero en (/ (+ 3 17) 4) a la función '/' se le está pasando un argumento numérico 4, pero también (+ 3 17), otra función con dos argumentos numéricos.

Esta es la esencia de un lenguaje de programación funcional y por eso decimos que LISP lo es. "Programación funcional significa: **escribir programas que operan a base de devolver valores en lugar de producir efectos colaterales**.

Estos efectos colaterales incluyen cambios destructivos en los objetos y la asignación de variables*. Una función destructiva es una que puede alterar los argumentos que se le pasan*.

Sólo unos pocos operadores LISP están pensados para producir efectos colaterales. En general, los operadores propios del lenguaje están pensados de manera tal que se invoquen para obtener los valores que devuelven. Nombres como sort (vl-sort), remove (vl-remove) o substitute (subst), que son palabras completamente entendibles.

Si se quieren efectos colaterales, es necesario utilizar **setq** sobre el valor devuelto. Esta misma regla sugiere que algunos efectos colaterales son inevitables.

Tener la programación funcional como ideal no implica que los programas nunca debieran tener efectos colaterales. Sólo quiere decir que no deben tener más de los necesarios.
Esta característica de la programación funcional no es arbitraria.

Los programadores LISP no adoptaron el estilo funcional por razones meramente estéticas. Lo usan porque facilita su trabajo. En el entorno dinámico de LISP, los

programas funcionales pueden ser escritos a una velocidad poco usual, y a la vez, pueden ser inusualmente confiables.

En LISP es comparativamente fácil el depurar los programas. Una gran cantidad de información se encuentra disponible en tiempo de ejecución, lo que ayuda en el rastreo de los errores. Pero aún más importante es la facilidad con la que pueden probarse los programas.

No es necesario el compilar el programa para probar su funcionamiento como un todo. Podemos probar las funciones individualmente, llamándolas desde el nivel superior del evaluador.

Esta comprobación de carácter incremental es tan valiosa que el estilo de programación LISP ha evolucionado para aprovecharla. Los programas escritos en un estilo funcional pueden ser comprendidos una función a la vez, y desde el punto de vista del lector, esta es su principal ventaja.

Sin embargo, el estilo funcional se adapta perfectamente a la comprobación incremental: los programas escritos en este estilo pueden ser también probados una función a la vez. Cuando una función ni examina ni altera el estado exterior, los errores se harán aparentes de inmediato. Una función así diseñada sólo puede afectar el mundo exterior a través de los valores que devuelve. En la medida que estos valores sean los esperados, podemos confiar en el código que los produjo.
Los programadores LISP experimentados de hecho diseñan sus programas de manera que puedan ser fácilmente probados:

Tratan de aislar los efectos colaterales en unas pocas funciones, de manera que la mayor parte del programa pueda ser escrito en un estilo puramente funcional.
Si una función debe producir efectos colaterales, tratan de que al menos posea una interfaz funcional.
Le dan a cada función un propósito único y bien definido
Cuando acaba de escribirse una función, pueden probarla sobre una selección de casos representativos, y una vez hecho esto pasar a la próxima función.

En LISP, como en cualquier otro lenguaje, el desarrollo se lleva a cabo en ciclos de escritura y comprobación. Pero en LISP el ciclo es muy corto: funciones aisladas, e incluso partes de funciones. Y si comprobamos todo a medida que lo escribimos, sabremos dónde buscar cuando se produzca un error: en lo último que se escribió.

CÓMO INSTALAR CLISP

Todo lo referente a la herramienta CLISP se encuentra en el directorio \\SIMUN\\c- simun\clisp\. Este directorio se encuentra dividido en dos subdirectorios:

- Clisp\documentati¢n: donde se encuentran todos los manuales necesarios para la utilización de CLISP, en formato pdf, estos manuales seencuentran también impresos en el laboratorio:

  - bpg.pdf: Basic programming Guide (volumen I).
  - apg.pdf: Advanced Programming Guide (Volumen II).
  - ig.pdf: Interfaces Guide (Volumen III).
  - usrguide.pdf: User's Guide.
  - abstract.pdf: Application Abstracts

- Clisp\executables: aqui se encuentra todo lo necesario para ejecutar la herramienta CLISP:

- Clisp\executables\clispwin\clispwin.exe: Clisp 6.01 Windows 95 interface.

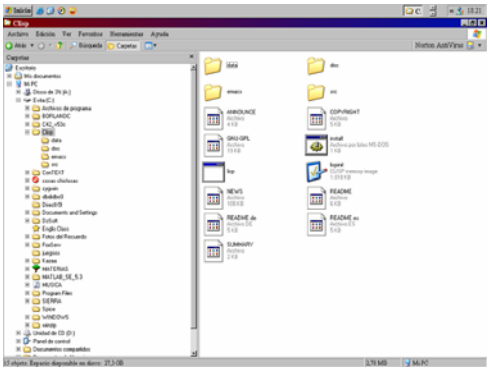
- Clisp\executables\clispedt\clispedt.exe: Clisp 6.01 Windows 95 Editor.

- Clisp\executables\clisphlp\clisphlp.exe: Archivo de ayuda para Windows Interface.

- Clisp\executables\clispdos\clispdos.exe: 32 Bit CLISP 6.01 DOS Executable.

- Clisp\executables\clispwrap\: Este directorio contiene las clases C++ queconstituyen el wraper necesario para la integración de Clisp en otra aplicación Windows.

- Clisp\executables\dll\: en este directorio se encuentran los archivos necesarios para integrar Clisp como una librería din mica en KSM.



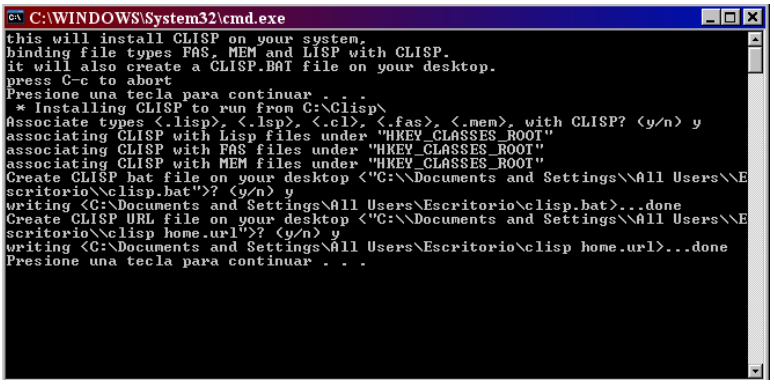
Los archivos necesarios para ejecutar la herramienta Clisp aisladamente son clispwin.exe, clispedt.exe, y clisphlp.exe. El ejecutable principal es el clispwin.exe, pero necesita que los otros dos ejecutables se encuentren en el mismo directorio para poder invocarlos desde el interfaz.

En el directorio Clisp existe un archivo Clisp.html que contiene la página web donde se encuentra todo tipo de información acerca de Clisp.

Para ser más específicos, la instalación de Clisp es Windows es realmente bien fácil: Lo necesario es primero tener una carpeta en el directorio Raíz, así:

Damos clic sobre el icono de Instalar 

y se presenta una pantalla así:



Luego se presiona cualquier tecla para continuar y ya se tiene listo Clisp para ser configurado el acceso directo.

Desde el sistema operativo Linux, para ejecutar el intérprete de Lisp CLISP se debe hacer desde una terminal. Para lanzar una terminal en el entorno KDE se puede hacer pulsando en el icono que se encuentra en la barra de herramientas. Una vez se ha lanzado la terminal se ejecuta el intérprete con el siguiente comando: **\$ clisp**

Y el resultado debe ser una pantalla similar a la de la figura 1. Esta pantalla es igual a la que se obtiene en entorno Windows. A partir de este momento se pueden empezar a introducir expresion LISP que serán interpretadas y se mostrará el resultado.

Solo si se va a instalar el intérprete en algún PC con Windows se deben seguir las siguientes instrucciones, de lo contrario se puede pasar directamente al siguiente apartado.

Normalmente, CLISP no se instala en el menú Inicio/Programas, por lo que lo primero será localizar el directorio donde se haya realizado la instalación que se genera al descomprimir el fichero empaquetado. Al hacerlo se generará algo como lo siguiente: D:\CLISP\clisp-2.27

Deberíamos encontrar el ejecutable lisp.exe. Si hacemos doble click sobre él habremos lanzado el shell del intérprete, posiblemente con un aviso del tipo:

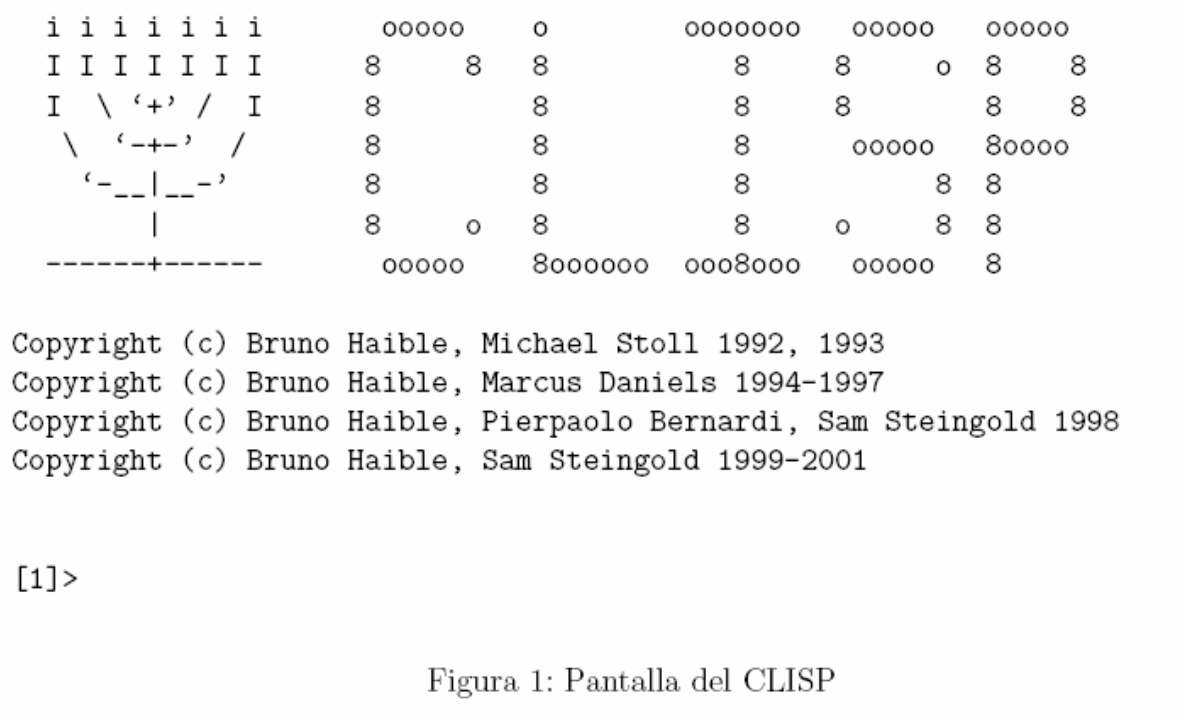


Figura 1: Pantalla del CLISP

WARNING: No initialisation file specified. Please try:
D:\CLISP\clisp-2.27\lisp.exe -M lispinit.mem
Sin corregir este problema no podremos trabajar. Lo más cómodo es crearnos un acceso directo que colocaremos en nuestro directorio de trabajo habitual o sobre nuestro escritorio. Para crear un acceso directo a CLISP seleccionaremos el fichero lisp.exe y pulsando el botón derecho del ratón accederemos a la opción Crear acceso directo.

Debemos editar las propiedades del acceso directo que acabamos de crear para completar la especificación de Destino como sigue:
D:\CLISP\clisp-2.27\lisp.exe -M lispinit.mem -B C:\apl\clisp-2.27

Si salvamos la configuración del acceso directo y hacemos doble click sobre el mismo, el intérprete de CLISP deberá lanzarse sin mayores contratiempos y sin alertas. Desdela línea de invocación se pueden configurar numerosos detalles del intérprete si se detalla el fichero de configuración a emplear como en el siguiente ejemplo (el directorio src de la distribución de CLISP contiene un ejemplo de fichero de configuración en config.lisp):
D:\CLisp\clisp-2.27\lisp.exe -M lispinit.mem -B C:\apl\clisp-2.27 -i C:\MyHome\config.lsp

El intérprete de LISP como su nombre indica, interpreta todas las entradas como expresiones LISP por lo que si en algún momento se produce algún error el intérprete pasa a modo depuración que se indica cambiando el prompt cambia incluyendo un número seguido de la palabra BREAK, como p.e.:

1. Break [23]>

El primer número (1) indica el nivel de depuración en el que se producido el error y la palabra Break indica que el depurador está activo. Por ahora nos limitaremos a saber cómo volver al \Top Level" o lo que es equivalente a abandonar el depurador.

Para ello escribiremos unwind + [ENTER], abort + [ENTER] o la más corta :a + [ENTER] tantas veces como sea necesario hasta conseguir que la palabra Break desaparezca del prompt.
1. Break [23]> :a
[9]>

En el depurador se pueden utilizar una serie de comandos para depurar el programa que ha producido. Algunos de estos comandos que pueden ser útiles en la depuración de un programa pueden ser:

:a Salir del depurador.
:h Muestra los comandos disponibles en el depurador.
:bt Muestra el estado de la pila.
:bt4 Muestra todos los marcos (frames) que el intérprete está intentando resolver.

CÓMO CARGAR EL PROGRAMA EN EL INTÉRPRETE

Para cargar ficheros en el intérprete existen múltiples posibilidades, pero por el momento expondremos aquí la más simple de todas, **basada en la función load**.

Supongamos: que deseamos cargar el fichero que acabamos de editar mi-programa.lisp. Si el mismo se encuentra en nuestro directorio de trabajo, escribiremos: **(load ``mi-programa.lisp'')**

y si todo ha ido bien, es decir, si el código no contiene errores, veremos un mensaje como:
;; Loading mi-programa.lisp ...
;; Loading of file mi-programa.lisp is finished.

Si nos encontramos en otro directorio podemos incluir la ruta relativa hasta el fichero anteponiéndola al nombre del fichero:**(load ``sources/mi-programa.lisp'')**

Si por el contrario queremos movernos a otra carpeta o directorio emplearemos la función cd: **(cd ``sources/'')**

Es importante que las rutas que pasamos como argumento a la función cd siempre terminen con la barra / si queremos evitar que el intérprete nos devuelva un error.

Por otra parte si queremos conocer nuestro directorio de trabajo actual podemos emplear la misma función cd sin argumentos: (cd) #P"C:\\Mis Documentos\\Lisp\\sources\\"

Por último, usaremos la función sin argumentos dir para listar el contenido del directorio de trabajo.

```
[1]> (dir)

C:\clisp\Mis-ejemplos\
C:\clisp\doc\
C:\clisp\src\
C:\clisp\absdiff.lisp                212       2001-10-04 12:25:50
C:\clisp\install.bat                 263       2001-06-08 23:02:56
C:\clisp\lisp.exe                 2215936      2001-07-16 18:51:36
C:\clisp\lispinit.mem              981828      2001-09-25 14:17:14
C:\clisp\malo.txt                      75      2001-11-30 08:56:50
C:\clisp\prueba.lisp                    7      2001-10-05 12:10:54
C:\clisp\prueba.lsp                   396      2001-11-30 09:16:52
```

CÓMO SALIR DE CLISP

Los comandos para cerrar el intérprete son:
(exit) , siempre se pone el paréntesis
o
(quit)

DESCRIPCIÓN

Invoca al intérprete y el compilador común de lisp. Invocado sin argumentos, ejecuta un lazo read-eval-print, en cuál las expresiones están en cambios uniformes de lectura desde la entrada, evaluado por el intérprete de lisp, y por su salida de resultados a la salida uniforme.

Invocado con -c, los archivos especificados de lisp son compilado a un bytecode que puede ser ejecutado más eficientemente.

OPCIONES

-h, **--help** Despliega un mensaje de ayuda de cómo usar clisp.

--version Despliega el número de versión de Clisp, es dado por la función CALL lisp-implementation-version).

--license Despliega un resumen de la información de licencia, el GNU GPL

-B *lisplibdir* Especifica el directorio de instalación. Este es el directorio que contiene el link o lazo hacia otros archivos de datos.

-M *memfile* Especifica la imagen de memoria inicial. Debe ser una memoria de descarga producido por la función saveinitmem.

-m *memsize* Pone la cantidad de pruebas de clisp de memoria en el startup. La cantidad puede ser dado como nnnnnnn (medido en byte), nnnn K o nnnn KB (medido en el kilobytes) o n M o MB n (medido en megaoctetos). La rebeldía es 2 megaoctetos. El argumento es forzado encima de 100 KB. -- Esta versión de clisp no es probable usar verdaderamente el memsize entero desde que la colección de basura reducirá periódicamente la cantidad de la memoria usada. Es por lo tanto común especificar 10 MB aunque sólo 2 MB será usado.

-L *language* Especifica el lenguaje usado por Clisp en la comunicación por el usario. Este lenguaje puede ser en inglés

-N *localedir* Especifica el directorio base de localidad de los archivos. Clisp buscará el mensaje en el catálogo localedir/language/LC_MESSAGER/clisp.mo

-E*domain encoding* Especifica la codificación usara por un dominio dado, haciendo caso omiso a cada una de los entornos de variables LC_ALL, LC_CTYPE, LANG. *domain* can be file, affecting *default-file-encoding*, or pathname, affecting *pathname-encoding*, or terminal, affecting *terminal-encoding*, or foreign, affecting *foreign-encoding*, or misc, affecting *misc-encoding*.

-q, **--quiet**, **--silent** Clisp despliega un banner en el startup y no envía mensaje de salida cuando termina

-w espera por la presión de cualquier tecla después de terminar un programa

-C Compilar cuando se carga el valor de las variables *load-compiling se usa con t. El código comienza a correrse y cuando es compilado luego se ejecuta. Este resultado se realiza de forma lenta, pero efectiva.

-norc Normally CLISP, carga el control de corrida del usario (RC) archivo en startup. El archivo cargado es _clisprc.lsp o clisprc.fas en el directorio raiz.

-i *initfile* ... especifica la inicialización de archivos a ser cargados en el startup. Estos pueden ser archivo lisp. Varias opciones –i puede dar todas las especificaciones de los archivos corridos en orden.

-c *lispfile* ... Compila laos lispfiles especificados por el bytecode (*.fas). La compilación de archivos instantáneamente cargados . Los archivos compilados entonces pueden ser cargado en vez de las fuentes y ganar de esa manera más eficiencia.

-o *outputfile* especifica la salida de archivos o directorios.

-l lista todos los archivos (*.list) de el programa o archivo que ha sido compilado o procesado.

-x *expressions* ejecuta una serie de expresiones en lugar de una lectura-evaluación-impresión en el ciclo de ejecución.

***lispfile* [*argument* ...]** Carga y ejecuta los lispfile.

(apropos *name*) Lista el símbolo que relaciona a *name*

ARCHIVOS

Main ejecutable

lispinit.mem inicializa una imagen de memoria

config.lsp configuración site-dependet.

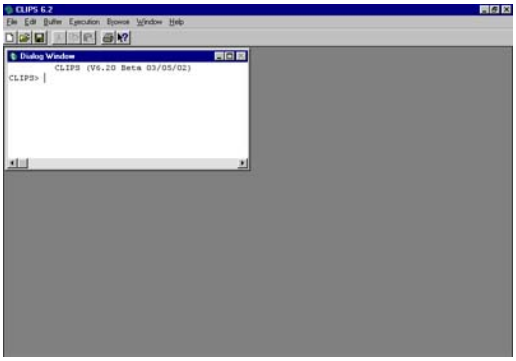
***.lsp** dirección lisp

***.fas** codigo lisp, compilado por clisp

***.lib** dirección lisp de información de librerías, generado y usado por el compilador de clisp.

***.c** Código C, compilado desde lisp por la dirección de clisp.

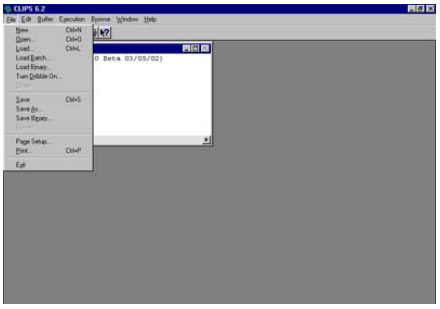
Ejecute el entorno de CLIPS desde el entorno Windows. Describa todas las ventanas que aparecen.



Al pulsar sobre el icono de CLIPS desde Windows nos aparece la siguiente ventana principal:

En ella sólo encontramos la ventana de diálogo abierta y CLIPS preparado para funcionar. A partir de aquí podemos cargar un fichero con reglas o ejecutar comandos directamente desde la consola y proceder a su ejecución.

Para cargar un fichero en el cual hayamos escrito previamente nuestro programa en CLIPS deberemos acudir al menú File, que se muestra en la siguiente figura:

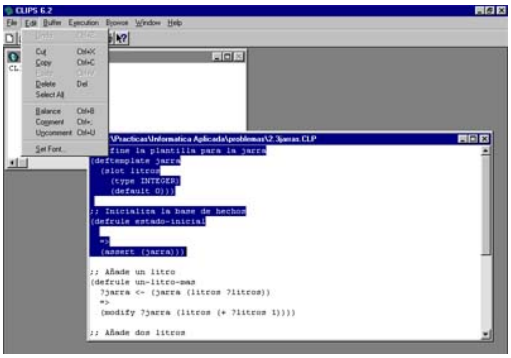


Para cargar nuestro fichero deberíamos pulsar la opción Load..., tras la cual se nos abre una ventana de diálogo en la cual eligiremos el fichero a cargar. También podemos cargar archivos por lotes (.BAT) o archivos binarios (.BIN) para su posterior ejecución.

El entorno CLIPS lleva incorporado un editor de textos, que nos permite trabajar con ficheros a la hora de crear nuestros programas de una manera muy sencilla. Las funciones de New, Open, Save, Save As y Save Binary nos resultan más que suficientes a la hora de crear, abrir, guardar, etc. ficheros de texto para escribir nuestro programa en CLIPS.

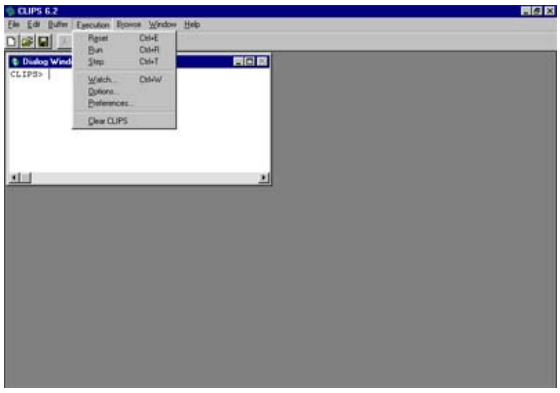
Para terminar, en el menú File encontramos también las opciones de configurar página (page Setup) e imprimir nuestros ficheros (Print), y por supuesto, la opción de salir del programa (Exit).

A continuación mostramos el menú Edit:



En este menú nos encontramos fundamentalmente con funciones de manejo de texto para la edición de nuestros programas en CLIPS. Cuenta con las funciones típicas de cualquier editor de textos: cortar, copiar, pegar, seleccionar todo, etc. y unas cuantas más de especial utilidad a la hora de programar en CLIPS: balance, que nos ayuda a ver si hemos escrito bien los paréntesis, y comment y uncomment, que añaden o eliminar un comentario de una línea de código. También podemos modificar el tamaño, tipo y color de las fuentes de nuestro archivo de texto.

Con el menú Buffer accedemos a otro tipo de funciones típicas en la editores de lenguajes de programación, con son las de encontrar (para encontrar una cadena en el texto), reemplazar, etc. Mientras este menú carece de dificultad, el siguiente merece especial atención: el menú Execution:



Este menú es de los más importantes, ya que con el nos ahorramos escribir repetidamente las órdenes más frecuentes a la hora de ejecutar y depurar un programa en CLIPS. Sus funciones son las siguientes:

* Reset: eliminar las reglas y hechos existentes y carga initial-facts y los hechos y reglas que hayamos especificado con Load...

* Run: ejecuta las reglas del programa CLIPS.

* Step: divide la ejecución en pasos. Por defecto, step vale uno, por tanto, cada vez que le demos veremos un ciclo de ejecución de CLIPS.

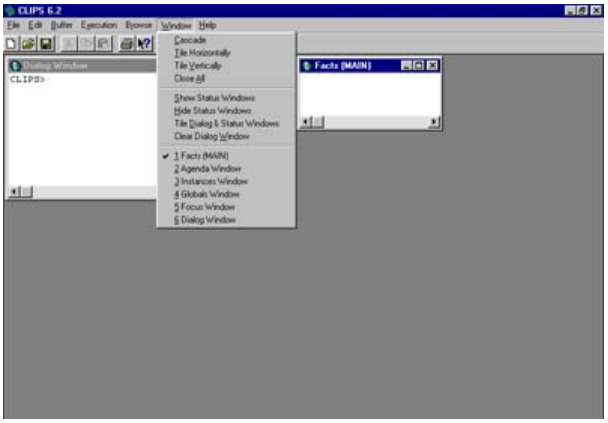
* Watch: nos permite ver la información seleccionada por pantalla conforme es añadida, modificada o borrada. Podemos elegir que nos muestre:la información de compilación, los hechos,las reglas, las funciones genéricas, etc.

*Options: cambia las opciones del entorno CLIPS.

* Preferences: nos permite elegir el incremento para cada paso ysi queremos que muestre las alertas.

* Clear CLIPS: pone CLIPS a cero, elimina todos los hechos y reglas existentes.

El siguiente menú resulta de gran utilidad cuando se ejecuta y depura un programa en CLIPS:

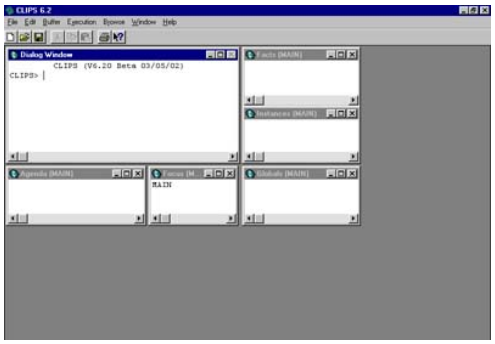


Mediante el primer conjunto de órdenes espicifcaremos a CLIPS como alinear las ventas en uso, con las típicas opciones: en cascada, horizontalmente, verticalmente o cerrarlas todas.

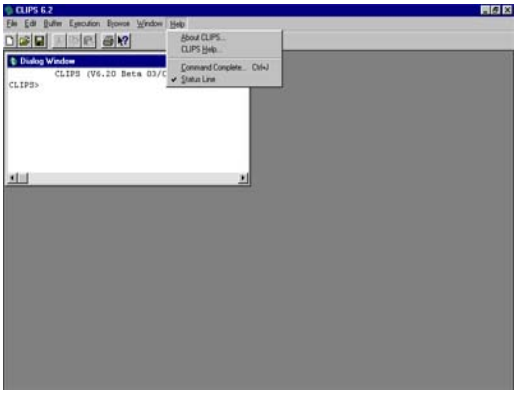
En el segundo conjunto nos ofrece una opción muy interesante: Show Startus Windows, que al elegirla nos muestra la ventana de la agenda, de hechos, de instancias, de globales y de focus. Mediante estas ventanas podemos observar como va evolucionando nuestro programa en CLIPS, por lo que son muy útiles a la hora de depurar programas. Mediante Hide Status Windows cerraremos todas las ventanas que abre Show Status Windows, mediante Title Dialog & Status Windows impediremos que estas ventanas abiertas se "overlap" y mediante Clear Dialog Window limpiaremos el texto que contenga la ventana de diálogo.

El último conjunto de órdenes nos permite elegir una a una cada una de las ventanas que dijimos abre Show Status Windows.

En la siguiente imagen se muestra como quedan dispuestas las ventanas al elegir Show Status Windows.



Por útlimo, sólo comentar brevemente el menú Help. Mediante el primer conjunto de órdenes se nos ofrece la posibilidad de consular la versión y fecha de creación de la misma de CLIPS que estamos utilizando (About CLIPS) y lanzar la ayuda de CLIPS (CLIPS Help) . El segundo conjunto nos permite activar/desactivar las funciones Command Complete, que nos muestra el posible comando que estamos escribiendo en CLIPS y Status Line, que muestra u oculta la barra de estado.



En LISP todo va entre paréntesis, al trabajar con un interprete se puede llamar a las diferentes funciones del programa que se estén usando y cambiar los valores de las variables cuando se necesita.

Para cargar un programa en el interprete se usa la instrucción **load**, la asignación se realiza con la instrucción **setq**.

Por ejemplo si se quiere usar el programa que implementa el espacio de versiones con el conjunto de datos **figure.da**t se tiene que hacer lo siguiente:

```
> (load "ml-util.lsp")
> (load "ver-spa.lsp")
> (setq *trace-vs* T)
> (load "figure.dat")
> (train-version-space *raw-examples*)
```

ESTRUCTURA DEL PROGRAMA

Una de Las características de LISP es la posibilidad de tratar las propias funciones como datos. En LISP, funciones e incluso programas enteros pueden ser utilizados directamente como entrada a otros programas o subrutinas.

Los componentes básicos al representar el conocimiento en un sistema basado en reglas como CLIPS son:

Memoria de trabajo: Contiene los datos que representan el estado, y sobre los cuales se van a hacer inferencias (aplicación de operadores).

Memoria de producción: Conjunto de reglas (operadores) que se expresan mediante sentencias de la forma: IF *precondición* - THEN *acción*

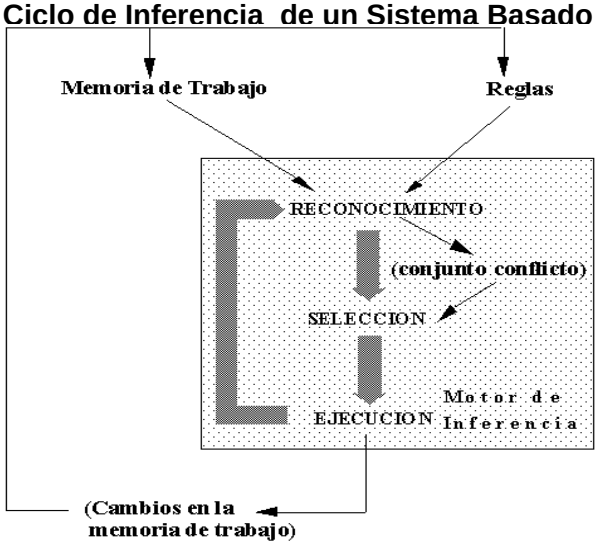
- La *precondición* es un conjunto de tests sobre los elementos de la memoria de trabajo. Los tests permiten reconocer el estado de la memoria de trabajo y determinar la posibilidad de aplicar una regla.
- La *acción* Especifica las modificaciones a realizar sobre la memoria de trabajo al aplicar la regla.

Un **motor de inferencia** que selecciona una regla entre las que se adecuan a la configuración de datos y la ejecuta. El ciclo de control usado por los motores de inferencia se denomina ciclo reconocimiento/actuación y consta de los siguientes pasos:

- ***Reconocimiento***. El motor de inferencia encuentra todas las reglas aplicables al contenido de la memoria de trabajo en el ciclo actual mediante la comparación de los elementos en la memoria de trabajo con la parte precondición de las reglas.

- ***Selección***. Sólo las reglas que son satisfechas por el contenido actual de la memoria de trabajo son candidatas a ser ejecutadas, denominándose al conjunto de estas reglas *conjunto conflicto*. El motor de inferencia selecciona una regla del conjunto conflicto para ser ejecutada en cada ciclo.
- ***Ejecución*.** El motor de inferencia ejecuta la parte acción de la regla seleccionada. La parte acción produce cambios en la memoria de trabajo cuando es ejecutada la regla.

Ciclo de Inferencia de un Sistema Basado en Reglas



El motor de inferencia ejecuta este ciclo repetidamente hasta que no haya reglas en el conjunto conflicto o se detenga el sistema explícitamente (predeterminando el número de ciclos que se van a ejecutar, o mediante una instrucción de parada en la parte acción de una regla).

REPRESENTACIÓN DE HECHOS

La memoria de trabajo contiene una lista de hechos, que son los datos con los cuales se puede razonar. Los datos de la memoria de trabajo definen el estado del proceso de razonamiento.

Cada hecho consta de un nombre de relación seguido de cero o más atributos con sus valores asociados. Por ejemplo para representar la información de una persona:

```
(persona (nombre "Vitorino Gómez Campo")
    (edad 32)
    (estudios Empresariales Derecho)
    (profesion abogado)
    (estado-civil soltero))
```

Antes de crear un hecho se debe definir una plantilla mediante **deftemplate**.

Deftemplate permite especificar si los atributos tendrán un único valor mediante **slot**, o pueden tener 0 o más valores mediante **multislot**. Además se puede añadir información adicional sobre los atributos como tipo de los atributos o restricciones sobre sus valores. .

La especificación del tipo de atributos puede ser:
VARIABLE, SYMBOL, STRING, LEXEME, INTEGER, FLOAT, o NUMBER.

VARIABLE? indica que el atributo puede contener cualquier tipo de dato;
LEXEME que el dato puede ser SYMBOL o STRING; y
NUMBER que el dato puede ser INTEGER o FLOAT?.

Se puede restringir los valores permitidos en un atributo mediante allowed-symbols, allowed-strings, allowed-lexemes, allowed-integers, allowed-floats, allowed-numbers, y allowed-values.

También es posible especificar un rango numérico mediante los valores mínimo y máximo mediante (range <mínimo> <máximo>), y la cardinalidad indicando el mínimo y máximo número de valores mediante (cardinality <mínimo> <máximo>) (?VARIABLE indica que no hay máximo o mínimo).

Si no se especifica el valor por defecto, se asume (default ?DERIVE), que especifica que como valor del atributo se derivará uno que cumpla con las restricciones impuestas. En el ejemplo de plantilla que se expone a continuación, si no se especifica otra cosa el atributo estado civil será soltero (el primero de la lista de valores permitidos).

Default también puede ir seguido de cualquier expresión cuyo valor será utilizado como valor por defecto; o seguido de ?NONE para indicar que no existe valor por defecto y que no se derivará ningún valor.

```
(deftemplate persona "Plantilla para describir persona"
  (slot nombre (type STRING))
  (slot edad (type INTEGER)
          (range 0 ?VARIABLE))
  (multislot estudios (type SYMBOL))
  (slot profesion (type SYMBOL))
  (slot estado_civil (type SYMBOL)
                (allowed-symbols soltero casado viudo  divorciado)))
```

Se ***denominan hechos ordenados*** a aquellos que no necesitan de plantilla para ser creados. Los hechos ordenados sólo tienen un atributo, y no precisan de atributo nombre. Son útiles en los siguientes casos:

Para representar flags. Por ejemplo (toda-orden-procesada) indica que todas las ordenes han sido procesadas.

Cuando los hechos contienen un único atributo, y el nombre del atributo es un sinónimo del nombre de la relación. Por ejemplo el hecho ordenado (tiempo 9:20) contiene tanta información como el hecho creado con plantilla (tiempo (valor 9:20)).

ADICIÓN, ELIMINACIÓN Y MODIFICACIÓN DE HECHOS

El estado de la memoria de trabajo puede ser alterado añadiendo nuevos hechos, y eliminando o modificando hechos de la lista. La instrucción (**fatcs**) visualiza la lista de hechos, y las siguientes instrucciones permiten modificar el estado de la memoria de trabajo:

- (**assert** <hecho>+). Se pueden asertar uno o más hechos. Por ejemplo, si queremos añadir un par de hechos utilizando la plantilla definida para personas:

```
CLIPS>
(assert (persona (nombre "Vitorino Gómez Campo")
                 (edad 32)
                 (estudios Empresariales Derecho)
                 (profesion abogado))
        (persona (nombre "Macarena López del Monte")
                 (edad 35)
                 (estudios Ingeniero)
                 (profesion empresario)))
<fact-0>
CLIPS> (facts)
f-0 (persona (nombre "Vitorino Gómez Campo")
             (edad 32) (estudios Empresariales Derecho)
             (profesion abogado) (estado_civil soltero))
f-1 (persona (nombre "Macarena López del Monte")
             (edad 35) (estudios Ingeniero)
             (profesion empresario) (estado_civil soltero))
For a total of 2 facts.
```

  Cada hecho tiene asociado un índice que puede ser utilizado para borrarlo o modificarlo.

- (**retract** <índice>+). Por ejemplo, (retract 0) elimina el hecho f-0.

```
CLIPS> (retract 0)
CLIPS> (facts)
f-1  (persona (nombre "Macarena López del Monte")
              (edad 35) (estudios Ingeniero)
              (profesion empresario) (estado_civil soltero))
For a total of 1 fact.
```

- (**modify** <índice> (<nombre-atributo> <valor>)+). **Modify** elimina el hecho modificado y aserta un nuevo hecho con el valor del o los atributos actualizados. Por ejemplo, si queremos actualizar el valor de la edad de Macarena a 36:

```
CLIPS> (modify 1 (edad 36))
<Fact-2>
CLIPS> (facts)
```

```
f-2  (persona (nombre "Macarena López del Monte")
          (edad 36) (estudios Ingeniero)
          (profesion empresario) (estado_civil soltero))
For a total of 1 fact.
```

- (**duplicate** <índice> (<nombre-atributo> <valor>)+). **Duplicate** actúa como modify añadiendo un nuevo hecho actualizado, pero no elimina el hecho modificado.

```
CLIPS> (duplicate 2 (nombre "Aparicia Perez del Rio"))
<Fact-3>
CLIPS> (facts)
f-2  (persona (nombre "Macarena López del Monte")
          (edad 36) (estudios Ingeniero)
          (profesion empresario) (estado_civil soltero))
f-3  (persona (nombre "Aparicia Perez del Rio")
          (edad 36) (estudios Ingeniero)
          (profesion empresario) (estado_civil soltero))
For a total of 2 facts.
```

Para asertar un conjunto de hechos que son ciertos antes de ejecutar el programa, por ej. Los hechos que definen el estado inicial, es útil la instrucción **deffacts**. Por ejemplo la siguiente instrucción define la información de toda la gente que hemos definido anteriormente:

```
(persona (nombre "Vitorino Gómez Campo")
      (edad 32)
      (estudios Empresariales Derecho)
      (profesion abogado))
(persona (nombre "Macarena López del Monte")
      (edad 35)
      (estudios Ingeniero)
      (profesion empresario))
(persona (nombre "Aparicia Perez del Rio")
      (edad 36)
      (estudios Ingeniero)
      (profesion empresario)
      (estado_civil soltero)))
```

Los hechos definidos con **deffacts** son asertados cuando se ejecuta la instrucción (**reset**). **Reset** elimina todos los hechos de la memoria de trabajo y aserta todos lo hechos definidos con instrucciones **deffacts**. Al iniciara CLIPS, automáticamente se ejecutan las siguientes instrucciones:

```
(deftemplate initial-fact)
(deffacts initial-fact (initial-fact))
```

Así que si ejecutamos reset con el grupo de hechos gente definido anteriormente la lista de hechos será la siguiente:

```
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1  (persona (nombre "Vitorino Gómez Campo") (edad 32)
          (estudios Empresariales Derecho)
          (profesion abogado) (estado_civil soltero))
f-2  (persona (nombre "Macarena López del Monte")
          (edad 35) (estudios Ingeniero)
          (profesion empresario) (estado_civil soltero))
f-3  (persona (nombre "Aparicia Perez del Rio")
          (edad 36) (estudios Ingeniero)
          (profesion empresario) (estado_civil soltero))
For a total of 4 facts
```

REGLAS

La memoria de producción está constituida por reglas. Las reglas representan el conocimiento sobre la resolución de problemas de un dominio específico incluyendo reglas físicas, conocimiento heurístico o reglas de experiencia.

Por ejemplo, consideremos los hechos y reglas que necesitaríamos para monitorizar y responder a posibles emergencias en un edificio. Informalmente podemos expresar el conocimiento mediante reglas de la siguiente forma:

SI la emergencia es incencio
ENTONCES la respuesta es activar los extintores de incendios.

Para convertir esta representación informal del conocimiento en reglas se deben definir primero los tipos de hechos a los que se refiere la regla. Una emergencia puede ser descrita mediante la siguiente plantilla:

```
(deftemplate emergencia (slot tipo)
     (allowed-symbols incendio inundacion
               cortocircuito sobrecarga))
```

Donde el atributo tipo puede contener los símbolos incendio, inundación, cortocircuito o sobrecarga. De la misma manera la respuesta puede representarse mediante la siguiente plantilla:

```
(deftemplate respuesta (slot accion))
```

Ahora podemos definir la regla en CLIPS:

```
(defrule emergencia-incendio "Un ejemplo de regla"
  (emergencia (tipo incendio))
 ==>
  (assert (respuesta (accion activa-extintor-incendios))))
```

Patrones en las reglas de producción

La parte **condición** de una regla, también denominada precondición, antecedente, o parte izquierda de la regla (Left-hand side, LHS), consta cero o más elementos condicionales (EC). El elemento más simple de EC es un simple patrón. Cada patrón consiste en una o más restricciones sobre el valor de alguno de los atributos de una hecho. En la regla emergencia-incendio, el patrón es (emergencia (tipo incendio)). La restricción sobre el atributo tipo indica que esta regla sólo será satisfecha por hechos creados con la plantilla emergencia que contengan el símbolo incendio en el atributo tipo.

CLIPS intenta encajar (reconocer) los patrones de las reglas con todos los hechos en la memoria de trabajo. Si todos los patrones de la regla reconocen hechos de la memoria de trabajo, la regla está activa y es puesta en la agenda. La agenda contiene la lista de reglas activadas. Es importante notar que una misma regla puede estar activada por combinaciones diferentes de hechos, por lo que puede aparecer varias veces en la agenda. A cada posible activación de una regla se la denomina instancia de la regla.

Sin una regla no tiene precondición se considera que tiene el patrón (initial-fact) implícitamente, de forma que toda regla sin precondiciones será activada al ejecutarse la instrucción (reset).

Reconocimiento de patrones

Como en otros lenguajes, CLIPS tiene variables para almacenar valores. Por convenio, todo identificador de una variable es precedido por el carácter ?, como por ejemplo ?color, ?nombre, ?valor.

Un primer uso de las variables es ligar una variable a un valor en la parte izquierda de una regla y usar el valor posteriormente en la parte derecha. Por ejemplo:

```
CLIPS> (clear)
CLIPS> (deftemplate personaje
   (slot nombre)
   (slot ojos)
   (slot pelo))
CLIPS>
(defrule busca-ojos-azules
  (personaje (nombre ?nombre) (ojos azules))
  =>
  (printout t ?nombre " tiene los ojos azules." crlf))
CLIPS>
(deffacts gente
(personaje (nombre Juan) (ojos azules) (pelo castagno))
(personaje (nombre Luis) (ojos verdes) (pelo rojo))
(personaje (nombre Pedro) (ojos azules) (pelo rubio))
(personaje (nombre Maria) (ojos castagnos) (pelo negro)))
CLIPS> (reset)
CLIPS> (run)
```

```
Pedro tiene los ojos azules.
Juan tiene los ojos azules.
```

Pedro y Juan tienen los ojos azules, por lo que la regla busca-ojos-azules es instanciada dos veces.

De paso, observa el uso de la función **printout** para escribir mensajes por la pantalla. **crlf** produce un salto de línea. La instrucción (run) ejecuta las reglas instanciadas.

El segundo uso de las variables consiste en utilizar la misma variable en múltiples lugares en la parte izquierda de una regla. La primera aparición de la variable liga un valor a dicha variable, y este valor es retenido para el resto de apariciones en la regla. De esta forma podemos definir relaciones entre los patrones de la parte izquierda de una regla, y definir restricciones adicionales.

Siguiendo con el ejemplo anterior, podemos asertar un hecho que indique que color de ojos buscar:

```
CLIPS> (undefrule *)
CLIPS>(deftemplate busca (slot ojos))
CLIPS>
(defrule busca-ojos
  (busca (ojos ?color-ojos))
  (personaje (nombre ?nombre) (ojos azules))
  =>
  (printout t ?nombre " tiene los ojos "
            ?color-ojos "."  crlf))
CLIPS> (reset)
CLIPS> (assert (busca (ojos azules)))
<Fact-5>
CLIPS> (run)
Pedro tiene los ojos azul.
Juan tiene los ojos azul.
```

Funciones avanzadas CLIPS para la correspondencia de patrones

CLIPS permite especificar patrones más sofisticados, así como funciones para manipular hechos en las reglas. Cabe destacar las siguientes posibilidades:

Ligar un hecho a una variable

La modificación, eliminación y duplicación de un hecho son operaciones frecuentes. El operador "**<-**" permite ligar un hecho a una variable. Un hecho ligado a una variable puede ser manipulado mediante las funciones modify, retract y duplicate. Por ejemplo:

```
CLIPS> (clear)
CLIPS> (deftemplate persona (slot nombre) (slot direccion))
CLIPS>  (deftemplate cambio (slot nombre) (slot direccion))
```

```
CLIPS> (defrule procesa-informacion-cambios
       ?h1 <- (cambio (nombre ?nombre) (direccion ?direccion))
       ?h2 <- (persona (nombre ?nombre))
      =>
      (retract ?h1)
      (modify ?h2 (direccion ?direccion)))
CLIPS>
(deffacts ejemplo
  (persona (nombre "Pato Donald") (direccion "Disneylandia"))
  (cambio (nombre"Pato Donald") (direccion "Port Aventura")))
CLIPS> (run)
CLIPS> (facts)
f-0  (initial-fact)
f-3  (persona(nombre "Pato Donald")(direccion "Port Aventura"))
For a total of 2 facts.
```

Comodines

Algunas veces es útil comprobar la existencia de un valor en un atributo sin necesidad de ligar el valor a ninguna variable. Para ello utilizaremos el símbolo "**?**", que puede ser visto como un comodín que reconoce cualquier valor. Por ejemplo:

```
CLIPS> (clear)
CLIPS>
(deftemplate persona (multislot nombre) (slot dni))
CLIPS>
(deffacts ejemplo
  (persona (nombre Jose L. Perez) (dni 22454322))
  (persona (nombre Juan Gomez) (dni 23443325)))
CLIPS>
(defrule imprime-dni
   (persona (nombre ? ? ?Apellido) (dni ?dni))
  =>
  (printout t ?dni " " ?Apellido crlf))
CLIPS> (reset)
CLIPS> (run)
22454322 Perez
```

La regla imprime-dni imprime los dni de las personas que tienen nombre compuesto y apellido, es decir aquellas que tengan exactamente tres elementos en el atributo nombre. Cuandose quiere un comodín que reconozca cero o más valores de un atributo multi-valor, se puede utilizar "**$?**". Por ejemplo, la siguiente regla imprimirá los dni de todas las personas con apellido, tengan o no nombre, o sea nombre sencillo o compuesto por dos o más.

```
CLIPS>
(defrule imprime-dni
   (persona (nombre $?nombre ?Apellido) (dni ?dni))
  =>
  (printout t ?dni " " ?nombre " " ?Apellido crlf))
```

```
CLIPS> (reset)
CLIPS> (run)
23443325 (Juan) Gomez
22454322 (Jose L.) Perez
```

Restricciones

Podemos representar patrones que reconozcan hechos que no tengan un determinado valor en un atributo, o que tengan alguno de los valores que se especifican. Los operadores operador **~**, **|** y **&** cumplen este propósito. Por ejemplo

```
(persona (nombre ?nombre) (pelo ~rubio))
```

reconocerá a toda persona con pelo que no sea rubio, y

```
(persona (nombre ?nombre) (pelo castagno | pelirrojo))
```

reconocerá a toda persona con el pelo castaño o pelirrojo.

El operador **&** se utiliza en realidad en combinación con los anteriores para ligar valor a una variable. Por ejemplo

```
(defrule pelo-castagno-o-rubio
  (persona (nombre ?nombre) (pelo  ?color&rubio|castagno))
   =>
  (printout t ?nombre " tiene el pelo " ?color crlf))
```

También es posible expresar predicados sobre el valor de un atributo directamente en el patrón. Por ejemplo:

```
(defrule mayor-de-edad
  (persona (nombre $?nombre) (edad ?edad&:(> ?edad 18))
   =>
  (printout t ?nombre " es mayor de edad." crlf))
```

Para indicar que el valor de un atributo debe ser igual al resultado de la evaluación de una función se indica mediante el operador =. Por ejemplo, para reconocer las personas con 2 años mas que la mayoría de edad se puede expresar mediante el patrón:

```
(persona (edad =(+ 18 2)))
```

Función test

Un elemento condicional puede evaluar una expresión, en lugar de representar un patrón que debe ser reconocido. Por ejemplo, para comprobar que la variable ?edad es mayor que 18 se puede representar mediante (**test** (> ?edad 18)).

Función bind

Si queremos ligar un valor a una variable para utilizarlo en varios sitios sin tener que recalcular el valor podemos hacerlo con la función **bind**. Por ejemplo:

```
(bind ?suma (+ ?a ?b))
```

Combinación de elementos condicionales con And, Or, y Not

Para que una regla este activada todos los elementos condicionales de la precondición deben reconocer hechos de la memoria de trabajo. Por lo tanto, hasta ahora todas la reglas vistas tienen un **and** implícito entre los elementos condicionales. CLIPS ofrece la posibilidad de representar **and** y **or** explícitos, así como la posibilidad de indicar mediante **not** que una regla será activada si no existe ningún hecho reconocido por un patrón.

En el siguiente ejemplo, en lugar de tener dos reglas que tienen la misma acción, se puede representar mediante una única regla que se activará si existe una emergencia de tipo inundación o si se han activado los aspersores del sistema de extinción de incendios.

```
(defrule desconectar-sistema-eléctrico
  (or (emergencia (tipo inundacion))
      (sistema-extincion-incendios (tipo aspersor-agua) (estado activado)))
  =>
  (printout t  "Desconectado sistema electrico." crlf))
```

La siguiente regla se activa en el caso de que no haya dos personas que cumplan los años el mismo día:

```
(defrule no-fecha-identica
  (not (and (persona (nombre ?nombre)
                     (cumpleagnos ?fecha)
            (persona (nombre ~?nombre)
                     (cumpleagnos ?fecha)))
  =>
  (printout t
   "No hay dos personas con el mismo cumpleagnos." crlf))
```

Búsqueda en profundidad y anchura

Para comprender como se representa un problema de búsqueda en un sistema basado en reglas como CLIPS, vamos a tratar el problema de las garrafas simplificado. El problemas consiste en llenar una garrafa de 3 litros con sólo dos operaciones, añadir un litro y añadir 2 litros. Utilizando la estrategia por defecto de CLIPS para ordenar la agenda, que es en profundidad, se presenta distintas etapas del desarrollo hasta llegar a un programa completo que realiza la búsqueda en anchura.

La plantilla utilizada para representar cada uno de los posibles **estados** de la garrafa es:

```
(deftemplate estado
            (slot garrafa
               (type INTEGER)
               (default 0)))
```

Para definir el **estado inicial** utilizaremos la siguiente regla:

```
; El estado inicial
(defrule Estado-Inicial
 ?inicial <- (initial-fact)
=>
(printout t "El estado inicial es la garrafa vacia"
          crlf)
 (assert (estado (garrafa 0))))
```

Las **reglas** que representan los operadores son:

```
;---------------------------------------------------------------
; Regla para agnadir un litro a la garrafa
(defrule Agnade-Un-Litro
   ?estado <- (estado
   (garrafa ?cantidad))
=>
   (printout t "Agnadiendo 1 litro a " ?cantidad " quedando "
            (+ 1 ?cantidad) " litros" crlf)
   (modify ?estado (garrafa (+ ?cantidad 1))))
;---------------------------------------------------------------
; Regla para agnadir dos litros a la garrafa
(defrule Agnade-Dos-Litros
   ?estado <- (estado
   (garrafa ?cantidad))
=>
   (printout t "Agnadiendo 2 litros a " ?cantidad " quedando "
            (+ 2 ?cantidad) " litros" crlf)
   (modify ?estado (garrafa (+ ?cantidad 2))))
```

La regla para detectar el **estado final** es:

```
;---------------------------------------------------------------
; Regla para detectar tarea realizada (Es decir la garrafa tiene 3 litros).
(defrule Hecho
   ?estado <- (estado
   (garrafa 3))
=>
   (printout t "Hecho - La garrafa tiene tres litros." crlf))
```

El fichero garrafa1.clp contiene las reglas anteriores. Para ejecutar el programa carga el fichero con la instrucción (load "garrafa1.clp"). A continuación ejecuta (reset) para que se creen los hechos definidos con deffacts, o bien se active la regla encargada de definir el estado inicial. La instrucción (run) puede ir seguida por el número de ciclos de inferencia que se quieren ejecutar. El resultado de la ejecución del programa es el siguiente:

```
CLIPS> (reset)
CLIPS> (run 1)
El estado inicial es la garrafa vacia
CLIPS> (run 1)
Agnadiendo 1 litro a 0 quedando 1 litros
CLIPS> (run 1)
Agnadiendo 1 litro a 1 quedando 2 litros
CLIPS>  (run 1)
Agnadiendo 1 litro a 2 quedando 3 litros
CLIPS>  (run 1)
Agnadiendo 1 litro a 3 quedando 4 litros
CLIPS>  (run 1)
Agnadiendo 1 litro a 4 quedando 5 litros
CLIPS>  (run 1)
Agnadiendo 1 litro a 5 quedando 6 litros
CLIPS>
```

Vuelve a ejecutar el programa paso a paso. Para ello debes ejecutar de nuevo (reset). Comprueba ahora el contenido de la agenda después de cada paso de ejecución. Podrás comprobar que, debido a la estrategia en profundidad, la regla instanciada con el hecho más reciente es colocada siempre en primer lugar. Por este motivo, no hay posibilidad de que se ejecuten las reglas Agnade-Dos-Litros y Hecho.

Encontrarás la solución a los problemas planteados anteriormente en el fichero garrafa2.clp. En este segundo programa se implementa la búsqueda en anchura.

Primero se asigna mayor prioridad a la regla Hecho, de forma que cuando ésta esté instanciada se coloque la primera en la agenda. Con esto conseguimos que el programa nos avise de que se ha alcanzado el objetivo. Pero como sigue habiendo reglas instanciadas no terminará la ejecución. Para solucionarlo se elimina el hecho estado de la memoria de trabajo:

```
;---------------------------------------------------------------
; Regla para detectar objetivo, es decir garrafa con 3 litros.
; Tiene la prioridad alta para que si es activada se ejecute.
;
(defrule Hecho
  (declare (salience 1000))
  ?estado <- (estado
             (garrafa 3))
=>
  (printout t "Hecho - La garrafa tiene tres litros." crlf)
      (retract ?estado))
```

Todavía no tenemos resuelto el problema. Si no hay ningún criterio para decidir entre dos instancias de reglas distintas se coloca antes en la agenda la que esta declarada primero. La regla Agnade-Dos-Litros no ha sido ejecutada en ningún momento. Si hubiesemos declarado esta regla antes que la regla Agnade-Un-Litro se hubiera ejecutado siempre la regla Agnade-Dos-Litros.

En este caso no habríamos encontrado la solución y el programa no se detendría nunca. Con este programa no se hace la búsqueda de forma adecuada ya que sólo tenemos un estado para expandir en cada paso, sólo podemos explorar una rama del árbol de búsquedas, y además no guardamos la historia de las acciones realizadas.

Podemos implementar la búsqueda en anchura manteniendo un hecho (profundidad ?) que indique el nivel de profundidad alcanzado en la búsqueda. De esta forma podemos expandir los hechos de la profundidad en curso para crear los del siguiente nivel. Todos los hechos del nivel en curso se expanden antes que los hechos del siguiente nivel. Esto es llevado a cabo por la regla actualiza-profundidad, la cual (debido a su baja prioridad) será llamada después de que todos los estados de un nivel han sido tratados. La regla incrementa el nivel de profundidad modificando el valor del hecho (profundidad ?). De esta forma se activan las reglas para expandir los nodos del siguiente nivel de profundidad.

La plantilla estado debe considerar el nivel de profundidad del nodo representado:

```
  (deftemplate estado
      (slot garrafa
      (type INTEGER)
      (default 0))
   (slot profundidad    ; Profundidad del nodo en el arbol
      (type INTEGER)
            (default 0)
            (range 0 ?VARIABLE)))
```

La regla que define el estado inicial crea el hecho profundidad, que inicialmente es cero, y el nodo inicial:

```
(defrule Estado-Inicial
  ?inicial <- (initial-fact)
 =>
  (printout t "El estado inicial es la garrafa vacia" crlf)
```

```
  (assert (profundidad 0))
  (assert (estado  (garrafa 0) (profundidad 0))))
```

Las reglas que definen los operadores se modifican de forma que se expandan los nodos del nivel de profundidad en curso antes de considerar los del siguiente nivel:

```
(defrule Agnade-Un-Litro
   (declare (salience 500))
   ?estado <- (estado (garrafa ?cantidad)
                      (profundidad ?profundidad))
   (profundidad ?nivel)
   (test (= ?profundidad ?nivel))
=>
  (printout t "Agnadiendo 1 litro a " ?cantidad " para llegar a "
            (+ 1 ?cantidad) " litros" crlf)
  (assert (estado (garrafa (+ ?cantidad 1))
            (profundidad (+ ?profundidad 1)))))
```

Para completar la estrategia de búsqueda en anchura se utiliza la regla actualiza-profundidad, que actualizará el nivel de profundidad cuando no queden más nodos que expandir del nivel en curso:

```
;---------------------------------------------------------------
;  Regla para actualizar la profundidad
;   Llamada solo si se han expandido todos los nodos de un nivel
;    (Por lo tanto prioridad mas baja)
;
(defrule actualiza-profundidad
   (declare (salience 100))
   ?profundidad <- (profundidad ?nivel)
=>
   (retract ?profundidad)
   (assert (profundidad (+ 1 ?nivel))))
```

Inicializa el sistema ejecutando (clear). A continuación carga el fichero garrafa2.clp, y comprueba su funcionamiento ejecutando paso a paso el programa y visualizando la agenda y la memoria de trabajo.

Para finalizar vamos a mejorar el programa de forma que escriba el camino de la solución encontrada. El programa encontrará todas las soluciones hasta el nivel de profundidad indicado por el hecho (maxima-profundidad ?). Se finalizará la ejecución si no quedan más nodos por expandir, o se sobrepasa el nivel de profundidad indicado. El programa garrafa3.clp realiza las tareas anteriores. En el programa se redefine la plantilla del estado, de forma que se guarda en un atributo el padre del nodo, y en el atributo operacion se registra la operación realizada para alcanzar el nodo desde su nodo padre. Observa que el hecho profundidad tiene dos valores, el primero registra la profundidad en curso, y el segundo el número de nodos expandidos hasta el nivel en curso.

Fíjate en la regla FIN, que detecta el fin de la búsqueda si no hay nuevos nodos que expandir, y en la regla Incrementa-cuenta-nodos, que actualiza la cuenta de nodos cada vez que es creado un nodo nuevo:

```
;*******************************
;*        PLANTILLAS           *
;*******************************
;---------------------------------------------------------------
; la cantidad de agua en la garrafa
;
(deftemplate estado
   (slot garrafa
        (type INTEGER)
        (default 0))
    (slot profundidad          ; Profundidad del nodo en el arbol
        (type INTEGER)
        (default 0)
        (range 0 ?VARIABLE))
   (slot nodo
        (type INTEGER)
        (default 0))          ; Numero de nodo en el arbol
   (slot padre                ; Nodo padre
        (type FACT-ADDRESS SYMBOL)
        (allowed-symbols no-padre))
   (slot operacion            ; Descripcion del movimiento
        (type STRING)))
;*******************************
;*      ESTADO INICIAL         *
;*******************************
;---------------------------------------------------------------
; El estado inicial
(defrule Estado-Inicial
  ?inicial <- (initial-fact)
 =>
  (printout t "El estado inicial es la garrafa vacia" crlf)
  (assert (profundidad 0 0))       ;0 profundidad  0 nodos
  (assert (estado  (garrafa 0) (profundidad 0) (padre no-padre)
          (operacion "No moviento")))
  (assert (maxima-profundidad 6))
  (assert (nodos 1)))
;*******************************
;*        OPERADORES           *
;*******************************
;---------------------------------------------------------------
; Regla para agnadir un litro a la garrafa
(defrule Agnade-Un-Litro
 (declare (salience 500))
 ?estado <- (estado
       (garrafa ?cantidad)
       (profundidad ?profundidad))
 (profundidad ?nivel ?)
```

```
 (test (= ?profundidad ?nivel))
 (test (< ?cantidad 3))
=>
 (assert (estado (garrafa (+ ?cantidad 1))
         (profundidad (+ ?profundidad 1))
         (padre ?estado)
         (operacion "Agnade un litro"))))
;---------------------------------------------------------------
; Regla para agnadir dos litros a la garrafa
;
(defrule Agnade-Dos-Litros
 (declare (salience 500))
 ?estado <- (estado
   (garrafa ?cantidad)
   (profundidad ?profundidad))
 (profundidad ?nivel ?)
 (test (= ?profundidad ?nivel))
 (test(< ?cantidad 3))
=>
 (assert (estado (garrafa (+ ?cantidad 2))
         (profundidad (+ ?profundidad 1))
         (padre ?estado)
         (operacion "Agnade dos litros"))))
;*****************************
;*  BUSQUEDA EN PROFUNDIDAD   *
;*****************************
;---------------------------------------------------------------
;  Regla para Increntar la cuenta de nodos
;  LLevamos la cuenta de nodos en profundidad y en nodos, de forma
;  que si no se crean nuevos nodos al aplicar los operadores
;  a los nodos del nivel de profundidad en curso pararemos la
;  busqueda
(defrule Incrementa-cuenta-nodos
    (declare (salience 1100))
    ?estado <- (estado (nodo ?estadoNodo))
    ?nodos <- (nodos ?cuentaNodos)
    (test (= ?estadoNodo 0))
 =>
    (assert (nodos (+ 1 ?cuentaNodos)))
    (modify ?estado (nodo ?cuentaNodos))
    (retract ?nodos))
;---------------------------------------------------------------
;  Regla para actualizar la profundidad
;  Llamada solo si se han expandido todos los nodos de un nivel
;    (Por lo tanto prioridad mas baja)
;
(defrule actualiza-profundidad
 (declare (salience 100))
 ?profundidad <- (profundidad ?nivel ?numNodosViejo)
 (nodos ?nodos)
 (test (> ?nodos ?numNodosViejo))
```

```
=>
 (retract ?profundidad)
 (assert (profundidad (+ 1 ?nivel) ?nodos)))
;*******************************
;*        ESTADO FINAL         *
;*******************************
;---------------------------------------------------------------
; Regla para detectar tarea realizada, es decir la garrafa tiene
; 3 litros. Tiene la prioridad mas alta para que cuando sea
; reconocida sea la que se ejecute.
;
(defrule Hecho
 (declare (salience 1000))
 ?estado <- (estado
   (garrafa 3))
=>
 (assert (completa ?estado))
 (assert (lista "Hecho")))
;*************************************
;* FIN BUSQUEDA E IMPRESION SOLUCION  *
;*************************************
;---------------------------------------------------------------
; Regla para construir el camino una vez encontrada una solucion
;
(defrule Completa
    (declare (salience 2000))
    ?estado  <- (estado (padre ?padre) (operacion ?movimiento))
    ?completa <- (completa ?estado)
    ?lista-vieja <- (lista $?resto)
=>
    (assert (lista ?movimiento ?resto))
    (assert (completa ?padre))
    (retract ?completa)
    (retract ?lista-vieja))
;---------------------------------------------------------------
; Regla para imprimir la solucion
;
(defrule Camino-Completado
    (declare (salience 2000))
    ?completa <- (completa no-padre)
    ?lista-vieja <- (lista ?primero $?movimientos ?ultimo)
=>
    (printout t "Solucion:" crlf ?movimientos crlf)
    (retract ?completa ?lista-vieja))
;---------------------------------------------------------------
; Regla para acabar cuando se alcanze cierto nivel de profundidad
;
(defrule alcanza-maxima-profundidad
    (declare (salience 1000))
    (profundidad ?nivel)
    (maxima-profundidad =(+ 1 ?nivel))
```

```
 =>
   (printout t crlf "Sobrepasado el nivel de profundidad ", ?nivel "." crlf)
   (halt))
;---------------------------------------------------------------
; Regla para acabar cuando no se agnaden mas nodos
;
(defrule FIN
   (declare (salience 100))
   ?profundidad <- (profundidad ?nivel ?numNodosViejos)
   (nodos ?numNodos)
   (test (= ?numNodos ?numNodosViejos))
=>
   (printout t "Un total de " (- ?numNodos 1) " nodos en el arbol"
           crlf)
   (retract ?profundidad)
     (halt))
```

CÓMO EDITAR UN PROGRAMA

Para editar un programa podemos emplear el editor que más nos guste ya que CLISP no incluye ninguno. Es recomendable que el editor que elijamos tenga dos características.

a) La primera es que **pueda \parear" paréntesis**, es decir que nos indique qué paréntesis a la derecha (cerramos cuando escribimos otro a la izquierda), lo que nos será de gran ayuda para corregir errores de edición; y la segunda que permita acceder a un determinado número de línea. Estas dos pequeñas \capacidades" pueden ser de enorme utilidad durante el proceso de edición de nuestros programas.

b) Desde dentro del intérprete se puede invocar al editor y se puede configurar para utilizar otro diferente al que viene por defecto (en Windows es el block de notas y en Linux es el vi). Aunque es mejor llamar al editor desde fuera del intérprete y de esta forma tener una ventana con el programa donde se edita y otra con el intérprete para ir comprobando el funcionamiento del programa. En la siguiente sección se indica como se puede cargar un programa en LISP desde el intérprete. Un elemento muy importante es que como en todos los lenguajes de programación, los ficheros fuente de LISP suelen tener una extensión estándar; esta extensión suele ser .lisp o .lsp.

Para llamar al editor desde dentro del intérprete se utiliza la siguiente función:
(ed ``mi-programa.lisp'')

Para cambiar el editor por defecto se debe redefinir el parámetro del intérprete *editor* de la siguiente forma: (defparameter *editor* ``notepad.exe'').