UNIVERSIDAD NACIONAL AUTONOMA DE NICARAGUA
UNAN-LEON

Tesis Para Optar al Titulo de Ingeniería en Sistemas de Información

Tema:

Soporte Para el Componente Curricular de Inteligencia Artificial y Sistemas Expertos basado en Lisp utilizando el editor Xanalys Lispworks 4.3

Integrantes:

- **Br. Allan Richard Reyes Alfaro.**
- **Br. William Antonio Rugama Escoto.**
- **Br. Axel Estefan Ruiz Vado.**

Tutor

Msc. Ricardo Espinoza

León 06 de Noviembre del 2007

Índice de Contenido

Marco Teórico

Unidad I: Inteligencia Artificial y Sistemas Expertos

Unidad II: Introducción a Lisp

Unidad III: Listas

Unidad IV: Definición de Procedimientos

Unidad V: Predicados y Estructuras de Control

Unidad VI: Iteración y Recursión

Unidad VII: Lectura/Escritura y Ficheros

PRESENTACIÓN

Somos un grupo de alumnos egresados del V año de la carrera de Ingeniería en Sistemas de Información de la UNAN-LEÓN.

Tenemos el agrado de presentarles la tesis sobre el tema "Soporte para el Componente Curricular de Inteligencia Artificial y Sistemas Expertos basados en LISP con el editor Xanalys LispWorks Personal Edition 4.3."

Confiamos que este trabajo tenga la aceptación y el agrado de todo el cuerpo docente y alumnos, y que sirva de orientación a las futuras generaciones de la universidad.

DEDICATORIA

- Dedicamos esta tesis en primera instancia a Dios que nos permitió guiarnos en el transcurso de nuestra vida y educación.

- También agradecemos a nuestros padres y familiares que con su apoyo y comprensión hicieron posible este logro en nuestra vida.

- A nuestros profesores por habernos brindado su esfuerzo, su amistad y apoyo en nuestro aprendizaje.

INTRODUCCIÓN

La asignatura de “Inteligencia Artificial y Sistemas Expertos” es impartida en el quinto año de la carrera de Ingeniería en Sistemas de Información por parte del Departamento de Computación de la Facultad de Ciencias en la UNAN-LEON siendo esta la importancia que tienen hoy en día el estudio y manipulación de Sistemas Inteligentes.

Para una mayor comprensión de este soporte, es esencial que el alumno posea algunos conocimientos en materia de programación (llamadas a funciones, rutinas o subrutinas, ficheros, etc.), algoritmos y estructuras de datos (por ejemplo como trabajan los procesos recursivos e implementación de los mismos), como también para observar similitudes y diferencias, ventajas y desventajas en los diferentes lenguajes, todos ellos impartidos en los semestres anteriores.

Con el desarrollo de este soporte para el componente curricular Inteligencia Artificial y Sistemas Expertos se pretende que el estudiante adquiera algunos conocimientos básicos sobre la teoría relacionada con dicho componente, inmiscuirse en la resolución de problemas, así como también, aprender a usar y realizar algunos ejercicios en lenguaje Lisp, muy utilizado para el desarrollo de aplicaciones en el estudio de la Inteligencia Artificial.

Con los conocimientos previos del alumno y la experiencia del profesor que imparta la materia, se espera que este soporte sea de gran utilidad en el transcurso que conlleva dicha materia.

ANTECEDENTES

La asignatura de Inteligencia Artificial y Sistemas Expertos esta formulada como una materia más en el pénsum académico del Departamento de Computación de la UNAN-LEON.

En años anteriores no se tenía una documentación, soporte o plan específico sobre los temas, capítulos o unidades que se deben impartir en la asignatura debido a que la universidad no cuenta con personal especializado para atender las generalidades con las que cuenta la materia.

Por este motivo los docentes encargados de enseñar la materia no contaban con un plan, diseño o soporte que les facilite el desarrollo de la asignatura en un orden satisfactorio, además, el cuerpo estudiantil no tenía un documento soporte o guía que les permita un mejor aprendizaje de la materia.

JUSTIFICACIÓN

Se ha elaborado este trabajo con el fin de proporcionar al cuerpo docente y estudiantil un documento que sirva de guía para el profesor que imparta la asignatura y como material de ayuda para el estudiante universitario.

Debido a que en la asignatura de Inteligencia Artificial y Sistemas Expertos no se tiene un plan específico de lo que se debe abordar y a la vez no se cuenta con personal especializado en la materia, este soporte, el cual fue elaborado con esmero y presenta de manera generalizada los temas fundamentales en el estudio de la clase, servirá como punto de inicio y referencia para una adecuada enseñanza y aprendizaje de la Inteligencia Artificial y sobre todo el lenguaje Lisp.

OBJETIVOS

General

- Desarrollar un soporte para el componente curricular “Inteligencia Artificial y Sistemas Expertos” que muestre algunos conceptos necesarios que el estudiante de Ingeniería en Sistemas de Información debe adquirir en el transcurso de dicha materia.

Específicos

- Explicar algunos términos generales y fundamentales en el estudio de la asignatura.

- Proporcionar un documento guía de ayuda para impartir la materia, y que sirva como elemento de orientación para futuros estudiantes y docentes.

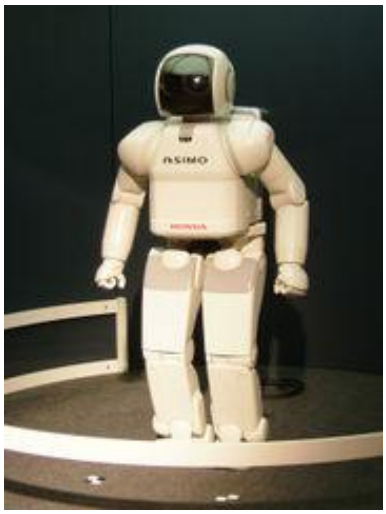
Unidad

1

INTELIGENCIA ARTIFICIAL Y SISTEMAS EXPERTOS

La primera unidad consiste en una descripción de lo que se llama Inteligencia Artificial y Sistemas Expertos, empezando por lo más básico como conceptos, características, ect. Una actual prueba que se debe realizar para saber que se esta frente a un sistema inteligente entre otras cosas.

Contenido:



Asimo, robot humanoide
creado por la compañía Honda.

1 Inteligencia Artificial

Definir la Inteligencia Artificial es una tarea realmente complicada. Desde que se menciono el término en la Conferencia de Darthmouth de 1956, se han propuesto muchas definiciones distintas que en ningún caso han logrado la aceptación sin reservas de toda la comunidad investigadora.

Al hablar de inteligencia artificial, tanto el término inteligencia como el adjetivo artificial plantean problemas.

Un problema es cuando una entidad desea algo y no conoce inmediatamente que acción o serie de acciones debe ejecutar para conseguirlo. Cualquier computador que resuelve un problema sin complicación alguna acude solamente a la aplicación de reglas consecutivas al pie de la letra llamado en *I.A.* como algoritmos.

Cuando hacemos uso de algoritmos para solucionar problemas se esta siendo inteligente, pero comparado con el ser humano y la presentación de problemas de distinta índole no existe algoritmo alguno que nos garantice las soluciones y/o opciones con que debe tratarse algún problema de nuestro diario vivir.

Al contar con un buen número de definiciones sobre la IA podemos elaborar una definición aproximadamente aceptable:

"Se denomina inteligencia artificial a la ciencia que intenta la creación de programas para máquinas que imiten el comportamiento y la comprensión humana."

Los hombres realizan procesos con poca computación pero con gran cantidad de conocimiento estructurado, en tanto que las computadoras realizan procesos fundamentalmente repetitivos utilizando bastante computación con datos muy estandarizados.

De hecho, la mayoría de los problemas humanos interesantes y difíciles no tienen soluciones algorítmicas practicables, ya que muchas tareas importantes originadas en contextos sociales, físicos, complejos, generalmente se resisten ante cualquier intento de descripción precisa y análisis riguroso.

En muchos problemas de inteligencia artificial los algoritmos de resolución exigen un tiempo de cómputo exponencial que sólo se mejora con soluciones heurísticas.

1.2 Características de la Inteligencia Artificial

a. Una característica fundamental que distingue a los métodos de *I.A.* de los métodos numéricos es el uso de símbolos no matemáticos.

b. El comportamiento de los programas no es descrito explícitamente por el algoritmo. La secuencia de pasos seguidos por el programa es influenciado por el problema particular presente.

c. El razonamiento basado en el conocimiento, implica que estos programas incorporan factores y relaciones del mundo real y del ámbito del conocimiento en que ellos operan.

d. La aplicación de los conocimientos dados y adquiridos de la *I.A.* nos deben permitir dar una solución a datos con problemas mal definidos construidos en el transcurso de su análisis.

1.3 Definición del Problema

El primer paso para diseñar un programa que resuelva un problema es crear una descripción formal y manejable del propio problema. Dado que por ahora no se conoce la forma de construir estos programas este proceso debe hacerse manualmente.

Hay problemas que por ser artificiales y estructurados son fáciles de especificar (por ej. el ajedrez, el problema de las jarras de agua, etc.). Otros problemas naturales, como por ej. la comprensión del lenguaje, no son tan sencillos de especificar.

Para producir una especificación formal de un problema se deben definir:

- Espacio de estados válidos.
- Estado inicial del problema.
- Estado objetivo o final.
- Reglas que se pueden aplicar para pasar de un estado a otro.

Un estado es la representación de un problema en un instante dado. Para definir el espacio de estados no es necesario hacer una enumeración de todos los estados válidos, sino que es posible definirlo de manera más general.

El estado inicial consiste en uno o varios estados en los que puede comenzar el problema. El estado objetivo consiste en uno o varios estados finales que se consideran solución aceptable. Las reglas describen las acciones u operadores que posibilitan un pasaje de estados.

1.4 Soluciones Heurísticas

Una Heurística: es una técnica que aumenta la eficiencia de un proceso de búsqueda. El objetivo es guiar al proceso de búsqueda en la dirección más provechosa sugiriendo el camino a seguir cuando hay más de una opción.

Las heurísticas pueden sacrificar la completitud, es decir, pueden pasar por alto una buena solución. Sin embargo, existen varios argumentos a favor de usarlas:

- Sin el uso de heurísticas se puede tener una explosión combinatoria.
- En muchos casos no se necesita la solución óptima sino una buena aproximación según Simón (1981), las personas resuelven problemas "satisfaciendo" y no optimizando".
- Las aproximaciones que se logran con heurísticas pueden ser malas para los peores casos de un problema, pero éstos raramente se dan en el mundo real.
- El esfuerzo de intentar comprender por qué funciona o no una heurística sirve para profundizar en la comprensión del problema.

En conclusión, las heurísticas representan el conocimiento general y específico del mundo, que hace que sea abordable solucionar problemas complejos.

1.5 Sistemas Inteligentes

Existen tres características que se le impondrá a un sistema como condición necesaria y suficiente para que sea inteligente en su comportamiento y tenga capacidad de aprender.

a. Considerar y proyectar el pasado: Se entiende por consideración del pasado al uso de ciertas técnicas que permitan un sistema beneficiarse de su experiencia. El resultado es que el sistema aprende incrementando de esta manera el horizonte de la prospectiva del futuro.

b. Prever el futuro: A la capacidad de realizar actos reflexivamente, es decir, teniendo en cuenta la posible consecuencia de la realización de dicho acto se la denomina previsión de futuro.

c. Intuición: Se conoce como una previsión de la verdad donde se encuentran los datos de la experiencia, es decir, lo que lo racional busca en la lógica aplicada al conocimiento, realizándolo en aquellas situaciones en las que dicha proyección no sea factible.

En otras palabras, es dotar al sistema de una capacidad de improvisación que le permita enfrentarse con un cierto grado de éxito aceptable ante situaciones insólitas adaptando la conducta a las circunstancias.

Esta exige el uso de funciones de valuación dinámica, es decir, aquellas capaces no sólo de modificar su coeficiente sino también sus argumentos de acuerdo con la situación en la que se encuentre en cada momento, en suma una evaluación dirigida por los datos.

1.6 Ciencias que Aportan a la Inteligencia Artificial

Como ocurre casi siempre en el caso de una ciencia recién creada, la *I.A* aborda tantas cuestiones confundibles en un nivel fundamental y conceptual que, adjunto a lo científico, es necesario hacer consideraciones desde el punto de vista de diversas ciencias.

De acuerdo a los conocimientos adquiridos en el paso de los años de cada una de estas ciencias su principal aportación a la *I.A.* es:

- **Filosofía:** Durante muchísimos años de filosofar (2000 aprox.), han venido surgiendo distintas teorías de que el razonamiento humano y el continuo aprendizaje se basa en un funcionamiento físico de la mente.

- **Matemáticas:** Facilitaron las herramientas para manipular las aseveraciones de certeza lógica así como las inciertas de tipo probabilista. Así mismo prepararon el terreno para el manejo del razonamiento con algoritmos.

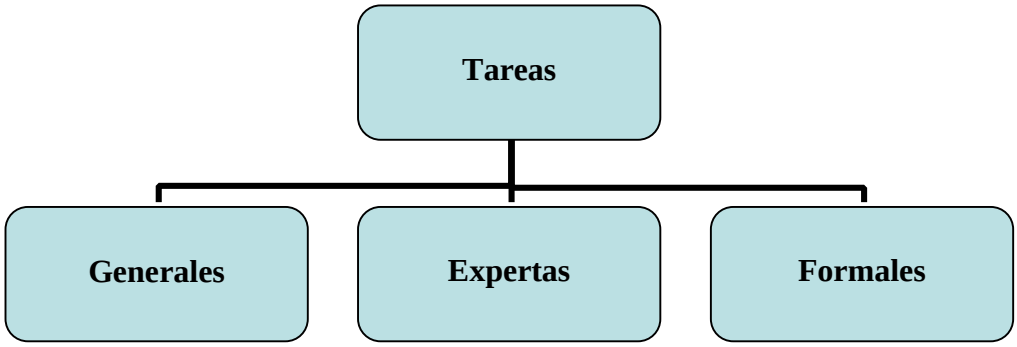
- **Psicología:** Hace uso de herramientas que nos permiten el estudio de la mente humana haciendo uso de un lenguaje específico para expresar las ideas y datos que se van obteniendo.

- **Lingüística:** Ofrece teorías sobre la estructura y el entendimiento de un lenguaje dado, demostrando que el uso de un lenguaje se ajusta dentro de este modelo.

- **Computación:** Ofreció el dispositivo que permite hacer realidad las aplicaciones de la inteligencia artificial.

1.7 Aplicaciones de la Inteligencia Artificial

Son muchas las tareas que se pueden derivar del estudio y la aplicación de la *I.A.*, en la vida son muchas las tareas que puede hacer una persona u otra, es por ello que las funciones que pueden realizar estos sistemas se clasifican en tres amplias tareas:



- Generales: Más difícil para una máquina que las tareas de un experto.
  - Percepción (visión y habla).
  - Lenguaje natural (comprensión, generación, traducción).
  - Sentido común.
  - Control de un robot.

- Tareas de los Expertos: Necesitan un conocimiento menor que el conocimiento necesario en las tareas más comunes.
  - Ingeniería (diseño, detección de fallos, planificación de manufacturación).
  - Análisis científico.
  - Diagnosis médica.
  - Análisis financiero.

- Tareas Formales.
  - Juegos (ajedrez, back gamón, damas).
  - Matemáticas (geometría, lógica, cálculo, demostración de propiedades).

1.8 Sistemas Expertos

Un *S.E.* es aquel capaz de almacenar el conocimiento de un experto en una especialidad determinada y limitada, pero a su vez de solucionar problemas mediante la inducción-deducción lógica.

Estos son intermediarios entre el experto humano, que transmite sus conocimientos al sistema, y el usuario de dicho sistema, que lo emplea para resolver los problemas que se le plantean.

Son sistemas de ejecución y transmisión del conocimiento. Así mismo, se definen mediante su arquitectura obteniendo una realidad palpable. Mientras que en las operaciones de programación clásicas se diferencia únicamente entre el propio programa y los datos.

1.8.1 Características de los Sistemas Expertos

Para que un sistema experto sea herramienta efectiva, los usuarios deben interactuar de una forma fácil, reuniendo dos características fundamentales que son:

1- Explicar sus razonamientos o base del conocimiento: los *S.E.* se deben realizar siguiendo ciertas reglas o pasos comprensibles de manera que se pueda generar la explicación para cada una de estas reglas, que a la vez se basan en hechos.

2- Adquisición de nuevos conocimientos o integrador del sistema: son mecanismos de razonamiento que sirven para modificar los conocimientos anteriores.

Se puede decir que los *S.E.* son el producto de investigaciones en el campo de la inteligencia artificial ya que esta no intenta sustituir a los expertos humanos, si no que se desea auxiliarlos para realizar más rápida y eficientemente todas las tareas que realiza con un grado de dificultad menor.

1.8.2 Estructura Básica de los Sistemas Expertos

En el caso de los *S.E.* se diferencian tres componentes principales. Son los siguientes:

a. Base de conocimientos (BC): Contiene conocimiento modelado extraído del diálogo con el experto.

b. Base de hechos (memoria de trabajo): Contiene los hechos sobre un problema que se ha descubierto durante el análisis a través de axiomas y reglas para hacer inferencias a partir de esos hechos acerca del dominio del sistema.

c. Motor de inferencia: Modela el proceso de razonamiento humano ejecutando procesos de inferencias, interpreta y evalúa los hechos en la base de conocimiento para proveer una respuesta. Que a su vez se divide en:

- Módulos de justificación: Explica el razonamiento utilizado por el sistema para llegar a una determinada conclusión.
- Interfaz de usuario: es la interacción entre el *S.E.* y el usuario y se realiza mediante el lenguaje natural.

1.8.3 Ventajas y Limitaciones de los Sistemas Expertos

Ventajas

- Permanencia: A diferencia de un experto humano un *S.E.* no envejece, y por tanto no sufre pérdida de facultades con el paso del tiempo.
- Duplicación: Una vez programado un *S.E.* lo podemos duplicar infinidad de veces.
- Rapidez: Un *S.E.* puede obtener información de una base de datos y realizar cálculos numéricos mucho más rápido que cualquier ser humano.
- Bajo costo: A pesar de que el costo inicial pueda ser elevado, gracias a la capacidad de duplicación el coste finalmente es bajo.
- Entornos peligrosos: Un *S.E.* puede trabajar en entornos peligrosos o dañinos para el ser humano.
- Fiabilidad: Los *S.E.* no se ven afectados por condiciones externas, un humano sí (cansancio, presión, etc.).

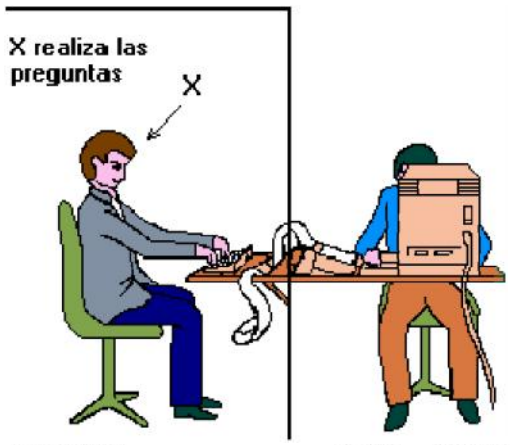
Limitaciones

- Sentido común: Para un *S.E.* no hay nada obvio. Por ejemplo, un sistema experto sobre medicina podría admitir que un hombre lleva 40 meses embarazados, se no se especifica que esto no es posible.
- Lenguaje natural: Con un experto humano podemos mantener una conversación informal mientras que con un S.E. no podemos.
- Capacidad de aprendizaje: Cualquier persona aprende con relativa facilidad de sus errores y de errores ajenos, que un S.E. haga esto es muy complicado.
- Perspectiva global: Un experto humano es capaz de distinguir cuales son las cuestiones relevantes de un problema y separarlas de cuestiones secundarias.
- Capacidad sensorial: Un *S.E.* carece de sentidos.
- Flexibilidad: Un humano es sumamente flexible a la hora de aceptar datos para la resolución de un problema.
- Conocimiento no estructurado: Un S.E. no es capaz de manejar conocimiento poco estructurado.

1.9 Actuar como Humano: La prueba de Turing

La prueba de Alan Turing propuesta en 1950, intenta ofrecer una definición de *I.A.* que se pueda evaluar. Para que un ser o máquina se considere inteligente debe lograr engañar a un evaluador, de que este ser o máquina se trata de un humano, evaluando todas las actividades de tipo cognoscitivo que puede realizar el ser humano.

Si el diálogo que ocurra y el número de errores en la solución dada se acerca al número de errores ocurridos en la comunicación con un ser humano, se podrá estimar según Turing que estamos ante una máquina "inteligente".



Aprobada si X no detectaba quién le respondía.

Hoy por hoy, el trabajo que entraña programar una computadora para pasar la prueba es considerable. La computadora debería se capaz de lo siguiente:

a. Procesar un lenguaje natural: Para así poder establecer comunicación satisfactoria, sea en español, inglés o en cualquier otro idioma humano.

b. Representar el conocimiento: Para guardar toda la información que se le haya dado antes o durante el interrogatorio.

c. Razonar automáticamente: Utiliza la información guardada al responder preguntas y obtener nuevas conclusiones o tomar decisiones.

d. Autoaprendizaje de la máquina: Con el propósito de adaptarse a nuevas circunstancias. El autoaprendizaje conlleva a la auto evaluación.

Para aprobar la prueba total de Turing, es necesario que la computadora esté dotada de:

- Vista: Capacidad de percibir el objeto que se encuentra en frente suyo.
- Robótica: Los robots son dispositivos compuestos de censores que reciben datos de entrada, una computadora que al recibir la información de entrada, ordena al robot que efectúe una determinada acción.

1.10 Lenguajes de Programación Involucrados

Los lenguajes de programación involucrados pueden ser nombrados como el principio de la programación en *I.A.* ya que estos lenguajes ofrecen características diseñadas especialmente para manejar problemas comúnmente encontrados en la *I.A.*
Por esta razón estos lenguajes son conocidos como lenguajes de *I.A.*

Dependiendo de las formas en como se estructuran las instrucciones se dividen en:

- Imperativos: PASCAL, C.
- Funcionales: Lisp.
- Declarativos: PROLOG, CHIP, OPS5.
- Orientados a Objetos: SmallTalk, HyperCard, CLOS.

Tradicionalmente Lisp y Prolog han sido los lenguajes que se han venido utilizando para la programación de sistemas expertos. Una de las características que estos dos comparten es que pueden ser utilizados para elaborar programas capaces de examinar otros programas, incluyéndose ellos mismos.

Esta capacidad es útil cuando se quiere que el programa explique sus conclusiones.
He aquí una breve introducción de los más usados:

Lisp

Su nombre se deriva de List Processor y fue el primer lenguaje de procesamiento simbólico desarrollado por John McCarty en 1958, al inicio era un lenguaje con el que los investigadores implementaban programas de computadoras capaces de razonar.

Se hizo muy popular por su capacidad de trabajar con símbolos, y fue elegido para el desarrollo de varios sistemas inteligentes. Actualmente, Lisp es utilizado en varios dominios que incluyen la escritura de compiladores, sistema para diseño mecánico asistido por computadora (Autocad), animaciones graficas y sistemas basados en conocimiento.

Prolog

Se deriva de Programing in Logic y es otro lenguaje de amplio uso en I.A, desarrollado en Francia en 1793 por Alain Colmenauer y su equipo. Al inicio fue usado para el procesamiento de lenguaje natural pero luego se hizo popular por su capacidad de trabajar con símbolos.

A partir de la década de los 80 tuvo una importante expansión ya que los japoneses decidieron usarlo para desarrollar sus sistemas de computación de quinta generación. Actualmente hay varios dialectos de Prolog para diferentes plataformas.

OPS5

Se deriva de Official Production System 5, y fue desarrollado por un equipo en la Universidad Carnegie Mellon. Es un lenguaje para ingeniería cognoscitiva que soporta la representación del conocimiento mediante reglas.

Incorpora un intérprete que incluye un mecanismo de encadenamiento progresivo y herramientas para edición y depuración de programas. Varias compañías han desarrollado aplicaciones comerciales de OPS5 para diferentes plataformas.

1.10.1 Algunas aplicaciones

Un agente, tal como se ha definido anteriormente, puede ser usado de múltiples maneras en el entorno empresarial actual, por ejemplo:

- **Newstracker:** Este programa recupera datos específicos. Los expertos definen a este sofisticado programa como un "asistente polivalente" de la primera generación.

- **Mind-it:** Este servicio gratuito de Internet envía un mensaje por correo electrónico cada vez que una página web (u otro documento) ha sido actualizado.

- **Eliza:** En 1966, Joseph Weizenbaum, del Instituto de Tecnología de Massachusetts, creó un programa para estudiar el lenguaje de comunicación entre el hombre y el computador. Fue programado para simular a un psicoterapeuta y contestar preguntas.

- **Express:** Este programa permite realizar múltiples búsquedas simultáneas en diferentes buscadores, y localizar información en Internet de manera fácil y rápida a través de una interfaz sencilla.

Unidad

2

INTRODUCCION A LISP 4.3

La segunda unidad de este soporte consiste en una breve introducción al Lisp 4.3, empezando por lo más básico como la utilización de las primitivas +,-,*, /. Así como aprender que Lisp 4.3 es de ejecución rápida, que multiplica la productividad y que resulta muy conveniente para diversas aplicaciones tanto, dentro y fuera de la inteligencia artificial.

Contenido:

2 INTRODUCCIÓN A LISP

Lisp llegó a ser fundamental como lenguaje de programación para las investigaciones de Inteligencia Artificial y sigue siendo uno de los más utilizados en este campo.

Su forma de programación es declarativa y no procedimental como en el caso de los famosos lenguajes de programación C/C++, Java, C#.NET, etc. En Lisp, los programas se construyen a partir de la composición de funciones.

Cuando se quiere implementar un problema en Lisp, éste se realiza escribiendo lo que se quiere conseguir, pero sin indicar paso a paso la secuencia de acciones que la computadora debe realizar.

Muchos programadores ya han usado esta metodología, ya que existen otros lenguajes declarativos como puedan ser el Prolog e incluso Sql. En estos lenguajes se especifica el qué se quiere obtener sin preocuparse del cómo.

2.1 Características del Lenguaje Lisp

a. Lisp posee la habilidad de expresar algoritmos recursivos que manipulen estructuras de datos dinámicos.

b. En Lisp, una función se expresa como una lista.

c. Lisp usa una sintaxis basada totalmente en paréntesis no teniendo necesidad de complejas reglas (como en los demás lenguajes de programación).

d. Lisp usa la notación prefija, la cual un operador debe escribirse antes de sus operandos. Por ejemplo: (+ 3 4).

e. Lisp es adecuado para la *I.A* ya que el código y los datos se tratan de la misma forma (como listas) siendo sencillo escribir programas capaces de escribir otros programas.

Lisp es un lenguaje capaz de trabajar con expresiones simbólicas (átomos o listas), todo en Lisp son expresiones simbólicas, desde la definición de funciones hasta el almacenamiento de los datos.

Se pueden distinguir diversos tipos de átomos, básicamente: los números (como el 4 del ejemplo), los strings literales, (como “¡Salud!”) y los símbolos (todos los demás del ejemplo).

En Lisp existen dos tipos básicos de palabras, los átomos y las listas. Todas las estructuras definidas posteriormente son basadas en estas palabras. Lisp 4.3 no hace distinción entre mayúsculas y minúsculas.

2.2 Lisp: Programación Funcional

En Lisp 4.3, funciones e incluso programas enteros pueden ser utilizados directamente como entrada a otros programas o subrutinas. 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".

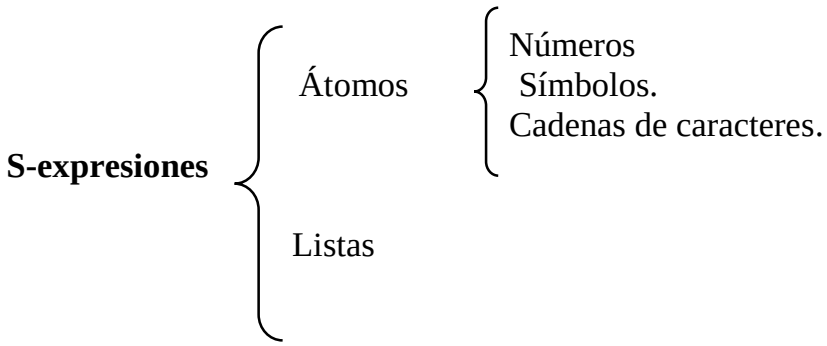
Estos efectos colaterales incluyen cambios destructivos en los objetos y la asignación de variables (con setq, por ejemplo).

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."

2.3 Los Objetos Básicos

Los objetos que se usan en Lisp se llaman S-expresiones (Simbolic expression) y estos objetos se clasifican en dos tipos básicos:



2.3.1 Átomos

Los átomos pueden ser palabras, tal como CASA, SACA, ATOMO, etc. o cualquier combinación como EDSDS, DFKM454, etc. En general, un átomo en Lisp puede ser cualquier combinación de las 26 letras del alfabeto (excluyendo obviamente la "ñ") en conjunto con los 10 dígitos.

Al igual que en otros sistemas, no son átomos aquellas combinaciones que comienzan con dígitos.
Ejemplos de átomos:

- Hola
- Casa
- Uno34
- F4fg5

2.3.2 Listas

El segundo tipo de palabras con las que trabaja Lisp son las listas. Una lista puede ser una secuencia de átomos separados por un espacio y encerrados por paréntesis redondos, incluyendo la posibilidad de que una lista contenga una sublista que cumple con las mismas características.
Ejemplos de listas:

- (ESTA ES UNA LISTA)
- (ESTALISTAESDISTINTAALAANTERIOR)
- (ESTA LISTA (TAMBIEN) ES DISTINTA)

2.4 Tipos de datos

Lisp ofrece una variedad de tipos de datos. Hay que destacar que en Lisp se trabaja con objetos que son de un determinado tipo, no con variables. Cualquier variable puede tomar como valor cualquier objeto Lisp. Es posible declarar explícitamente que una variable va a contener un determinado tipo de objeto, aunque dicha declaración se puede omitir y el programa funcionaría correctamente.

Dicha declaración sirve únicamente como información, se puede utilizar el predicado typep para averiguar si un objeto pertenece a un determinado tipo. Además, la función type-of retorna un tipo al cual pertenece el objeto.

A continuación mencionaremos algunos de los tipos de datos más destacables que son:

- **Números:** Hay varios tipos de números: números enteros, con una representación limitada por el tamaño de la memoria. números racionales, números en coma flotante así como números complejos.

- **Caracteres:** Representan letras, dígitos, etc. Se pueden construir cadenas de caracteres (strings).

- **Símbolos:** Son objetos con un valor asociado (a veces se denominan símbolos atómicos). Para hacer referencia a un símbolo (para comprobar o cambiar su valor) basta con usar su nombre. Los símbolos también sirven para dar nombre a funciones y a variables en los programas.

- **Listas:** Son secuencias construidas como celdas enlazadas, denominadas celdas cons. Hay un objeto especial que denota la lista vacía y que es Nil. Todas las demás listas se construyen recursivamente añadiendo un nuevo elemento al principio de una lista ya creada.

- **Funciones:** Son objetos que se pueden invocar como procedimientos, pueden llevar argumentos y retornan un valor (o varios).

- **Otros:** Son los packages, pathnames, streams, random-states, condiciones excepciones), clases, métodos, etc.

2.5 Comandos Fundamentales

1. Quote: Devuelve lo mismo que recibe, es decir, cuando quieras especificar una expresión pero no quieras que esta sea evaluada. La utilidad de este comando aparece cuando se utiliza, por ejemplo, en el comando Car entre paréntesis.
Por ejemplo:

(quote(devuelve lo mismo))	→ (devuelve lo mismo)
(quote((hi)(world)cool))	→ ((hi)(world)cool)
(quote(())	→ ()

(CAR(A B C)) en este caso, Car buscará el primer elemento de la lista que genere la función A, pero como A no es una función (a menos que se defina como tal) generará un error. La sentencia correcta sería: (CAR(QUOTE(A B C))

Un error común es escribir algo así: (CAR(QUOTE ERROR)) ya que en este caso QUOTE retorna el átomo ERROR, y CAR debe recibir como argumento una lista.
Como QUOTE es empleada frecuentemente, existe una abreviatura de este ” ‘ ”.

2. Car: Devuelve el primer elemento de una lista, si la lista esta vacía devuelve NIL.
Por ejemplo:

(CAR (‘((rich) will axl))) → (rich)
(CAR (‘((tesis)))) → (Tesis)
(CAR (‘(()(monografia)))) → ()

3. Cdr: Devuelve una lista formada por los elementos de esta misma, excepto el primero, si la lista esta vacía devuelve NIL.
Por ejemplo:

(CDR(‘(esta es una prueba))) → (es una prueba)
(CDR(‘((esta es)una prueba))) → (una prueba)
(CDR(‘((esta es una prueba)))) → ()
(CDR(‘(()(esta es una prueba)))) → ((esta es una prueba))

4. Cons: Construye y retorna una lista con los argumentos pasados a esta misma.
Por ejemplo:

(CONS(‘(esta es)quote(unaprueba))) → ((esta es)una prueba)
(CONS(quote esta)(quote())) → (esta)

5. Atom: Identifica todo aquello que no es una lista no vacía devuelve T si el argumento es un átomo, NIL en otro caso.
Por ejemplo:

(ATOM(‘ abc54)) → T
(ATOM(‘(un ejemplo))) → **N**il
(ATOM ‘abc54) → T
(ATOM ‘estoesunejemplo) → Nil

6. Eq: Devuelve T si el primer argumento y el segundo argumento son el mismo símbolo y NIL en caso contrario.
Por ejemplo:

(EQ 'hola 'hola) → T
(EQ ‘hola ‘b) → Nil
(EQ(quote hola)(quote hola)) → T
(EQ(quote g)(quote hola)) → Nil

7. Null: Devuelve cierto si su argumento es la lista vacía, y falso en otro caso. Esta operación es la misma que realiza la función Not, si bien Not suele utilizarse para invertir el valor booleano de una expresión, Null suele utilizarse para comprobar si una lista esta vacía.

Por ejemplo:

(NULL ()) → T
(NULL ‘esta es una prueba) → Nil
(NULL (quote())) → T

2.6 Evaluación de las Expresiones en Lisp

A menudo nos interesará conocer el valor de una expresión. Dicho valor es calculado automáticamente por el intérprete por medio de un proceso denominado evaluación.

Para ello existe una regla muy sencilla que es:

1- Si la expresión es un átomo, el resultado de su evaluación depende de si es un número o un símbolo.

- Si es un número, el resultado de su evaluación es él mismo.
- Si es un símbolo, el intérprete busca su valor asociado (es decir, lo trata como si fuera una variable y devuelve su contenido). Si el símbolo no tiene ningún valor asociado, su evaluación dará error.

2- Si es una lista, el primer elemento de la lista es interpretado como la función a aplicar para obtener su valor, y el resto de elementos son los argumentos que se le pasan a dicha función.

- Si la función no está definida, su evaluación dará error.
- Si está definida, primero se evalúan todos los argumentos y después se realiza la llamada a la función con los resultados de evaluar los argumentos.

2.7 Valores Lógicos

T: es un símbolo y su valor es T y representa lo verdadero. Este símbolo puede evaluarse a así mismo (T→ T).

NIL: es un símbolo y su valor es NIL y representa lo falso. Este puede escribirse también de esta forma “( )” que significa lista vacía. Al igual que T, puede evaluarse a si mismo (NIL→ NIL).

En los símbolos Lisp no distingue mayúsculas de minúsculas. Por ejemplo Max, Min, SqRt, ExPT, son entendidas por Lisp.

Unidad 3

LISTAS

En esta unidad se abordara el concepto de lista, el mas importante en Lisp 4.3, porque todos los programas en Lisp 4.3 son en sí mismos, listas. Estudiaremos también el manejo de las listas, así como algunos operadores fundamentales para el tratamiento de estas, conociendo las funciones más importantes en el transcurso de dicha unidad.

Contenido:

3 LISTAS

Definición: Una lista se define como 0 o mas expresiones, encerrados entre paréntesis. Cada una de estas expresiones se denomina elemento de la lista. Los elementos pueden ser de cualquier tipo de dato valido en Lisp 4.3, incluidas las listas. El concepto de lista es el mas importante en Lisp 4.3, ya que todos los programas en Lisp 4.3 son, en sı mismos, listas.
Por ejemplo:

( ) → ninguno
(A B C) → los tres símbolos A B C.
(A (B C)) → el símbolo A y la lista (B C).

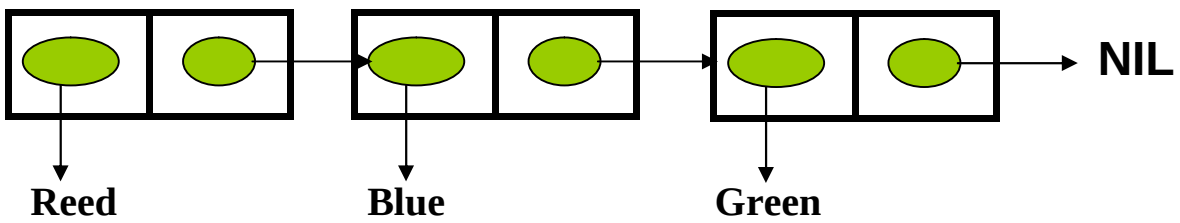
En Lisp, una lista se reconoce porque va entre paréntesis, en cambio, un átomo no.

3.1 Representación

Tienen dos formas de representación:

1- **Impresa:** Es más compacta y fácil de escribir. Una lista es un conjunto de elementos (elementos de la lista) encerrados entre paréntesis.
Por ejemplo:
(RED BLUE GREEN)
(ABCDEF)

2- **Interna:** Hace referencia a la forma en que esta implementada en la memoria del ordenador. En la memoria las listas se organizan en celdas. Las celdas se unen mediante punteros, y tienen dos posiciones que pueden contener información o punteros a otras celdas.



3.2 Características de las Listas

1- **Listas Vacías o Nulas:** NIL o "( )" es el elemento especial que es a la vez un átomo y una lista.

2- **Listas no vacías:** La más sencilla es la que tiene un único elemento. Cualquier lista no vacía de n elementos se puede ver como una estructura en la que se repite n veces la estructura de la lista con un único elemento.
La lista elemental de un solo elemento, tiene dos tipos de información:
   a. El dato correspondiente a esa posición de la lista.
   b. El dato correspondiente al siguiente elemento de la lista.

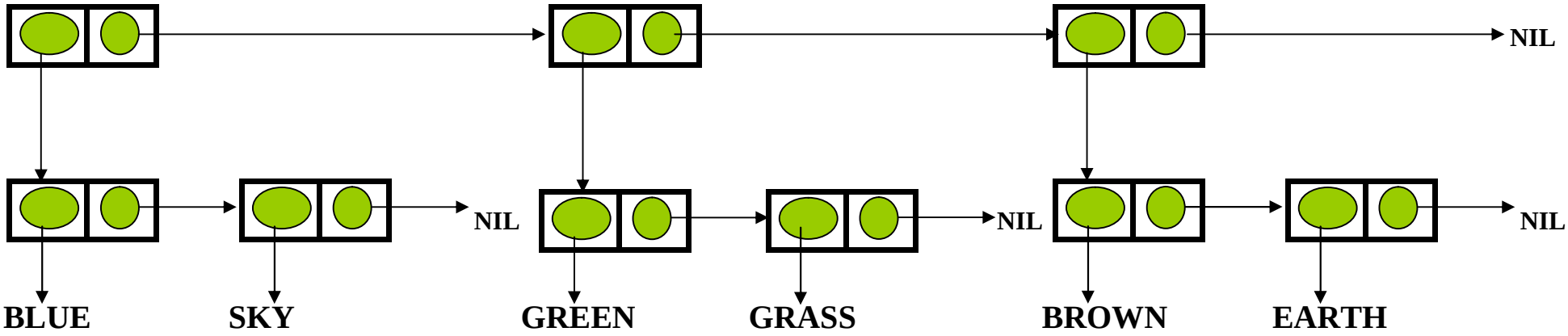
3- **Listas anidadas:** listas que contienen otras listas

4- **Longitud de las listas:** Numero de elementos que tiene.
En listas anidadas: es el número de elementos que aparecen en el primer nivel de paréntesis (en la representación impresa) o el número de celdas del nivel superior (en la representación interna)
Por ejemplo:

(LENGTH ' (A (B C) D)) 3

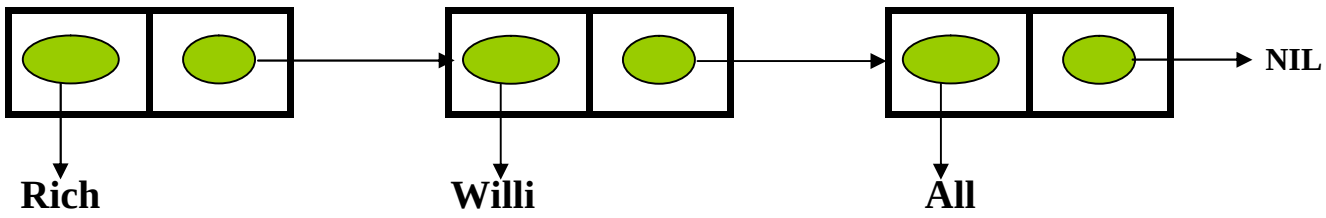
Representación Impresa: ((BLUE SKY)(GREEN GRASS)(BROWN EARTH))

Representación Interna:

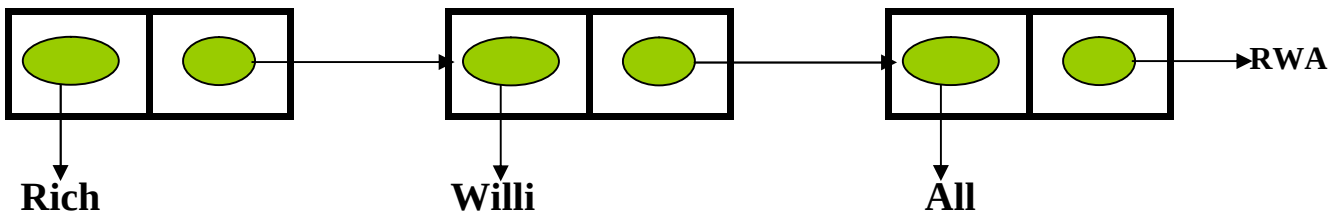


3.3 Estructuración Interna de las Listas

1- **Lista Propia:** Secuencia de celdas terminando con un NIL, el nombre viene del hecho de que cuando la lista es de más de dos elementos, se suele emplear una lista propia porque es más fácil de manejar debido a que NIL marca siempre el final de la lista.
Por ejemplo: **(Rich Willi All)**



2- **Lista Impropia:** Secuencia de celdas que no terminan con un NIL.
Por ejemplo: (**Rich Willi All. RWA)**



Nótese que como es una lista "diferente" se emplea una notación diferente, de ahí que se señale que la lista es impropia con un punto entre los dos elementos.

Existe una regla muy sencilla para distinguir una lista Propia de una Impropia:

- La lista propia tiene tantos conses como elementos.
- La lista impropia de n elementos tiene n-1 conses.

El punto de la lista impropia debe estar separado de ambos átomos. Por ejemplo, **(1.2)** es una lista propia con un número real, mientras que **(1 . 2)** es una lista impropia con dos números enteros. Con los símbolos ocurre lo mismo, **A.B** es un sólo símbolo y no dos, puesto que el punto se puede emplear en los nombres de los símbolos.

3.4 Construir Listas

1- Cons: Una celda cons es como un registro con dos campos, denominados CAR y CDR. Las celdas cons se usan primordialmente para representar listas, de forma que una lista se define como la lista vacía o una celda cons cuyo CDR es una lista.

(CONS s l)

Para organizar una lista simple, necesitamos de al menos dos cosas: los átomos que la forman y algún método de enlace entre ellos.
Por ejemplo:

(CONS ’a ’b) → (A . B)
(CONS ’a (CONS ’b (CONS ’c ’()))) → (A B C)
(CONS ’a ’(b c d)) → (A B C D)

2- List: Hace una nueva lista añadiendo arbitrariamente un número de entradas y construyendo un conjunto de celdas que terminan en NIL.

(LIST EXP1 ... EXPN)

Es decir construye y retorna una lista con sus argumentos, cuyos elementos son los valores de las expresiones EXP1, ..., EXPN. Esta sentencia se utiliza para crear listas propias.
Por ejemplo:

(LIST 'A 'B 'C) → (A B C)
(LIST 'A 1 'B 2) → (A 1 B 2)
(LIST '* (+ 2 3) '(/ X 2)) → (* 5 (/ X 2))
(LIST NIL) → (NIL)

3- List*: Es como list excepto que la última celda cons de la lista que se construye tiene como CDR el último argumento que se pase a la función.

(LIST* EXP1 ... EXPN)

Es decir construye y retorna una lista con sus argumentos, cuyos elementos son los valores de las expresiones EXP1, ..., EXPN, solo que el final del penúltimo argumento es el último argumento, o sea se utiliza para crear lista impropias.
Por ejemplo:

(LIST* ’a ’b ’c ’d) → (A B C . D)
(LIST* ’a ’b’c’(d e f)) → (A B C D E F)

4- Append: Añade listas juntas copiando su primera entrada y haciendo que el CDR de la ultima celda de la copia apunte a la segunda entrada.

(APPEND L1 ... LN)

O sea, devuelve una copia de la concatenación de las listas L1, ... , LN permitiendo crear una nueva lista. Mientras que LIST y CONS aceptan listas y átomos como argumentos, APPEND sólo acepta listas, excepto en el último argumento.

Por ejemplo:

(APPEND ’(a b c) ’ (d e f) ’() ’(g)) → (A B C D E F G)
(APPEND '(A B) '(C D)) → (A B C D)
(APPEND NIL '(A B)) → (A B)

3.5 Acceder a los Elementos de una Lista

1. Car: Devuelve el primer elemento de L, si L es una lista no vacía y NIL, en otro caso.

(CAR L)

Su equivalente es (FIRST L) que también devuelve el primer elemento de L.
Por ejemplo:

(CAR '(A B C)) → A
(CAR ()) → NIL
(CAR 'A) → ERROR
(FIRST '(A B C)) → A
(FIRST ()) → NIL

2. Cdr: Devuelve una lista formada por los elementos de L, excepto el primero, si L es una lista no vacía y NIL, en otro caso.

(CDR L)

Su equivalente es (REST L) que también devuelve la lista formada por los elementos de L, excepto el primero.
Por ejemplo:

(CDR '(A B C)) → (B C)
(CDR ()) → NIL
(CDR 'A) → ERROR
(REST '(A B C)) → (B C)
(REST ()) → NIL

3. Second, Third y Last

(SECOND L): Devuelve el segundo elemento de L.
(THIRD L): Devuelve el tercer elemento de L.
(LAST S): Devuelve una lista formada por el último elemento de S, sea o no S una lista.
Por ejemplo:

(SECOND '(A B C D)) → B
(THIRD '(A B C D)) → C
(LAST '(A B C)) → (C)
(LAST 23) → 23

4. Nth, Nthcdr

(NTH N L): Devuelve el N-ésimo elemento de L, empezando a contar por 0.
(NTHCDR N L): devuelve el N-ésimo CDR de L, empezando a contar por 0.

Por ejemplo:

(NTHCDR 1 '(A B C)) → (B C)
(NTH 1 '(A B C)) → B
(SETQ L '(A (B C) D E)) → (A (B C) D E)
(NTH 0 L) → A
(NTH 2 L) → D
(NTH 8 L) → NIL
(NTHCDR 0 L) → (A (B C) D E)
(NTHCDR 2 L) → (D E)
(NTHCDR 8 L) → NIL

5. Length (LENGTH L): Devuelve el número de elementos de L.

(LENGTH 'A B C D)) → 4

Subseq (SUBSEQ L ind num): Esta función toma dos o tres argumentos, el primero es la lista, el segundo un entero indicando la posición donde comenzar, y el tercero, si existe, es un entero indicando la posición donde detenerse.

Por ejemplo:

(SUBSEQ '(a b c d) 1 3) → (B C)
(SUBSEQ '(a b c d) 1) → (B C D)

3.6 Clasificación de las Listas

3.6.1 Listas Asociativas: Una lista asociativa o a-lista es una lista de pares, donde cada par es una asociación.

El CAR de un par se denomina clave y el CDR es el dato o sea una lista de pares punteados. Tiene la siguiente estructura:

((CLAVE-1 . VALOR-1) ... (CLAVE-N . VALOR-N))

Es decir, cada elemento de una A–lista es un par formado por la CLAVE (el CAR) y el VALOR (el CDR).

Las A–listas pueden crearse mediante SETQ.

Por ejemplo:

(SETQ L '((B . 2)(C . 3))) → ((B . 2) (C . 3))

3.6.1.1 Funciones sobre Listas A-Lista

- **Acons:** Añade el elemento compuesto por la CLAVE y el VALOR al comienzo de la lista AL y devuelve la lista así formada.

(ACONS CLAVE VALOR AL)

Es decir, construye una nueva lista asociativa añadiendo el par (clave valor) a la lista que se le pasa como tercer argumento.
Por ejemplo:

(ACONS 'A 1 '((B . 2)(C . 3))) → ((A . 1)(B . 2)(C . 3))

- **Pairlis:** Forma una A–lista mediante las claves de L1 y los valores de L2 y se la añade a la cabeza de AL.

(PAIRLIS L1 L2 AL)

Por ejemplo:

(PAIRLIS '(A B C) '(1 2 3) ()) → ((C . 3) (B . 2) (A . 1))
(PAIRLIS '(A B) '(1 2) '((C . 3))) → ((B . 2) (A . 1) (C . 3))
(PAIRLIS '(A B) '((1) (2)) ()) → ((B 2) (A 1))

- **Assoc:** Devuelve el elemento de la A–lista AL cuya clave es SIMB.

(ASSOC SIMB AL)

Por ejemplo:

(ASSOC 'B '((A . 1) (B . 2) (C . 3))) → (B . 2)
(ASSOC '(B) '((A . 1) ((B) 1) (C D))) → Nil

- **Rassoc:** Devuelve el primer elemento de la A–lista AL cuyo valor es VAL.

(RASSOC VAL AL)

Por ejemplo:

(RASSOC 1 '((A) (B . 1) (C D E))) → (B . 1)
(RASSOC '(D E) '((A) ((B) 1) (C D E))) →NIL
(RASSOC '(D E) '((A) ((B) 1) (C D E)) :TEST #'EQUAL) →(C D E)

- **Sublis:** Devuelve una copia de la expresión S en la que todas las ocurrencias de las claves de la A–lista AL se han sustituido por sus valores.

(SUBLIS AL S)

Por ejemplo:

(SETQ DICCIONARIO '((2 . DOS) (4 . CUATRO) (+ . MAS)(= . IGUAL-A)))
→ ((2 . DOS)(4 . CUATRO)(+ . MAS)(= . IGUAL-A))

(SUBLIS DICCIONARIO '(2 + 2 = 4))
→ (DOS MAS DOS IGUAL-A CUATRO)

3.6.2 Lista de Propiedades: Donde los p son átomos que denotan propiedades y las V son valores asociados a estas propiedades. Tiene la siguiente estructura:
(p1 V1 p2 V2 .. pn Vn)
Donde los indicadores p1…pn y los valores v1…vn son S–expresiones.
Las P–listas pueden crearse mediante SETF.

(SETF (SYMBOL-PLIST PL) L)

Asocia a PL la P–lista L y devuelve PL. Setf es distinto de Setq, ya que solo así podrán actuar las funciones sobre P–listas.
Por ejemplo:

(SETF (SYMBOL-PLIST 'PEPE) '(EDAD 40 HIJOS (PEPITO PEPITA)))
→(EDAD 40 HIJOS (PEPITO PEPITA))

3.6.2.1 Funciones sobre Listas P-Lista

- **Get:** Se utiliza para obtener información de una lista de propiedades usamos la función "get" como sigue:

  (GET Nombre p)

  "Nombre" identifica a la p-lista y p identifica la propiedad cuyo valor se desea.
  Por ejemplo:

  (GET 'PEPE 'EDAD) → 40
  (GET 'PEPE 'HIJOS) → (PEPITO PEPITA)
  (GET 'PEPE 'PADRES) → NIL

- **Put:** Se utiliza para reemplazar información en una lista de propiedades.

  (PUT Nombre p v)

  "Nombre " identifica a la p-lista, p la propiedad cuyo valor va a ser reemplazado y v es el nuevo valor.
  Por ejemplo:

  (PUT 'PEPE 'HIJOS 'Raulito)
  (PUT 'PEPE 'EDAD 50)

- **Symbol:** Devuelve la P–lista asociada a PL, si la hay y NIL, si no.

  (SYMBOL-PLIST PL)

  Por ejemplo:

  (SYMBOL-PLIST 'PEPE) → (EDAD 40 HIJOS (PEPITO PEPITA))
  (SYMBOL-PLIST 'JUAN) → NIL

- **Remprop:** Esta función quita una propiedad y su valor asociado de la p-lista.

  (REMPROP Nombre p)

  "Nombre" identifica la lista de propiedades afectada y p identifica la propiedad y el valor que han de quitarse.
  Por ejemplo:

  (REMPROP 'PEPE 'EDAD) → T
  (SYMBOL-PLIST 'PEPE) → (MADRE ANA HIJOS (PEPITO PEPITA))

Por tanto, las listas de propiedades son muy útiles para definir lo que se conoce en otros lenguajes como "registros".Típicamente una lista de propiedades representa un nodo en una lista mayor, enlazada de registros, todos con el mismo conjunto de propiedades.

Esta lista mayor es conocida comúnmente como un "archivo" en otros lenguajes. Por ejemplo, el registro PEPE puesto anteriormente puede ser un nodo de una lista que contenga todas sus características en una determinada etapa de su vida.

3.7 Copiar Listas

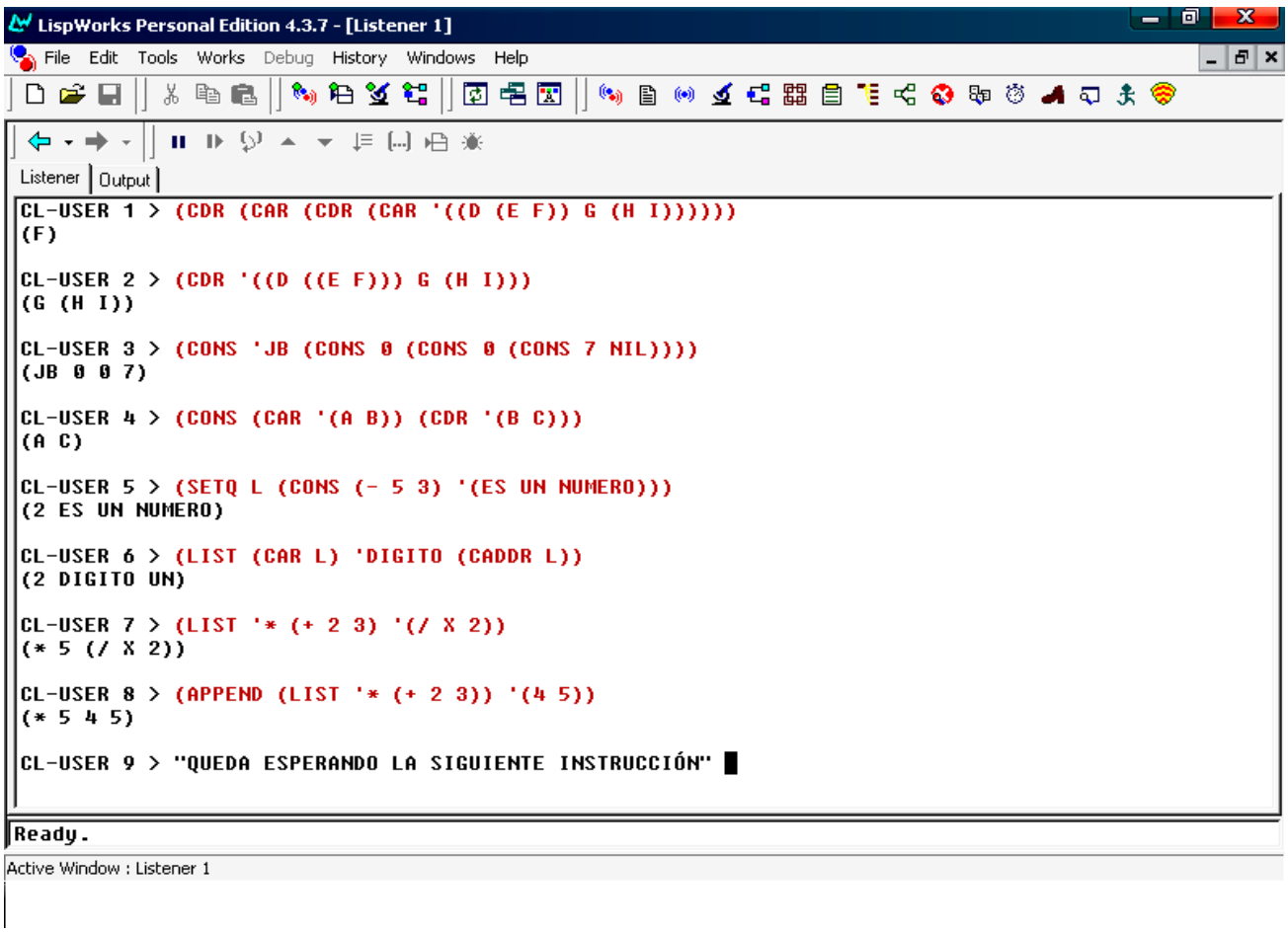
Las siguientes funciones sirven para copiar listas:

- **Copy-List:** Copia listas de un sólo nivel de anidamiento.
- **Copy-Alist:** Sirve para copiar listas de asociación.
- **Copy-Tree:** Copia recursivamente todos los conses del árbol.

En el caso de COPY-LIST y COPY-ALIST, si hay listas anidadas no se realiza una copia de las mismas, pero sí quedan enlazadas con la lista principal.

La diferencia estriba en que si se realiza una operación destructiva sobre alguna parte de la lista no copiada (sólo enlazada), esa modificación se realiza tanto en el original como en la copia.

3.8 Ejemplos Prácticos:



Unidad

4

DEFINICION DE FUNCIONES

La presente unidad trata de explicar la manera como se pueden definir procedimientos de usuario a partir de las primitivas del lenguaje, componiéndose estos mismos de llamadas a otras primitivas y otros procedimientos previamente definidos.

Contenido:

4 DEFINICIÓN DE FUNCIONES

Un programa grande suele dividirse en una serie de pequeñas formas (una expresión simbólica en posición de ser evaluada) o funciones de usuario más fáciles de implementar y depurar. Estas llamadas tendrán la forma de listas que podrán anidarse unas dentro de otras de acuerdo a lo que requiera la complejidad de la manipulación que quiera realizarse de los datos aportados como argumentos.

Las funciones forman el bloque básico de construcción en Lisp 4.3. Una función es un tipo de objeto, independientemente de cualquier símbolo o nombre de símbolo que se le pueda asociar. Las llamadas a funciones tendrán forma de listas las cuales se podrán anidar unas con otras dependiendo de la complejidad del programa a crear y de su correspondiente manipulación.

4.1 Tipos de Funciones

1. Función Destructiva: es una que puede alterar los argumentos que se le pasan, no realizan una copia sino que operan sobre las celdas cons de los argumentos. El cómo pueden modificar los argumentos es dependiente de la implementación. Por ejemplo: Nconc, Nreverse.

2. Función No Destructiva: realizan una copia de los argumentos cuando es necesario devolver el resultado esperado sin modificar ningún argumento. Por ejemplo: Append, Reverse.

Sólo unos pocos operadores Lisp están pensados para producir efectos colaterales. Lisp sigue una filosofía de tratamiento no-destructivo de los parámetros, de modo que la mayoría de las funciones devuelven una lista resultado de efectuar alguna transformación sobre la que recibieron, pero sin alterar esta última.

4.2 Funciones con Nombre

La manera más simple de definir una función es por medio de la macro **DEFUN**, que toma como argumentos un símbolo, una lista de argumentos y un cuerpo.
Es llamada función macro porque esta no evalúa sus argumentos como las demás funciones, sino que los toma de manera literal.

Evaluará sus argumentos una vez que esta sea invocada sustituyendo sus argumentos por los valores que se quieren procesar. Posee la estructura siguiente:

(DEFUN <nombre> <lista de parámetros> <documentación> <cuerpo>)

Significado de las palabras claves:

- Nombre: es el nombre de la función definida.

- Lista de parámetros: es la lista de parámetros (argumentos), son variables locales que no afectan a posibles valores previos, en general. Si no hay argumentos, es obligatorio poner ( ).

- Documentación: la documentación es opcional, aunque ya veremos más adelante que es sumamente útil.

- Cuerpo: las formas que constituyen el cuerpo de la función se ejecutan como si estuviesen encerradas dentro de progn.

El cuerpo esta implícitamente dentro de un bloque cuyo nombre es el mismo que el de la función, por lo que se puede utilizar return para salir de la ejecución de la función.
Por ejemplo:

(DEFUN CUADRADO (N) (* N N)) →CUADRADO
(CUADRADO 3) → 9

(DEFUN HIWorld () (PROG ((Res)) (SETQ Res (LIST 'hola 'mundo)) (RETURN Res))) →HIWORLD

Una vez que la función esta definida, solo hay que llamarla como a cualquier otra función, poniendo entre paréntesis su nombre y argumentos si los tiene.

(HIWorld) → (HOLA MUNDO)

En la lista de argumentos se pueden poner palabras clave que tienen significados especiales. De momento, sólo vamos a nombrar la palabra clave &OPTIONAL que significa que todos los argumentos a continuación de ella son Opcionales.
Por ejemplo:

```
(DEFUN Potencia (x &optional (n 2))
    (if (= n 0)
       1
       (* x (Potencia x (- n 1)))))      → POTENCIA
(Potencia 5)                             → 25
(Potencia 5 0)                           → 1
```

Se puede especificar el nombre del argumento y su valor inicial en una lista, como está en el ejemplo, o se puede poner solamente el nombre del argumento (sin paréntesis) en cuyo caso el valor por defecto es NIL.

Como Defun es una macro, y no una función, se comporta de un modo diferente y no evalúa todos sus argumentos. Tanto el símbolo como los argumentos se toman literalmente y no es necesario utilizar quote.

Los parámetros de un procedimiento son variables ligadas a el y los cuales se les asigna valor al iniciar la ejecución del procedimiento. Es preciso aunque delicado, decir que al iniciar la ejecución de un procedimiento una variable se liga y se le asigna cierto valor. Por consiguiente es común afirmar que la variable esta ligada a su valor. Así se ha creado una nueva función Lisp que puede llamarse desde el nivel superior o anidada en otras funciones.

4.3 Funciones Anónimas

Algunas funciones se usan pocas veces, por lo tanto, no tiene mucho sentido crear una función con nombre mediante Defun.

Para describir que se hace sin definir un procedimiento con nombre se usa una expresión lambda, expresión similar a la forma Defun. Solo es cuestión de reemplazar Defun por Lambda y omitir el nombre del procedimiento. Es una lista con la sintaxis siguiente:

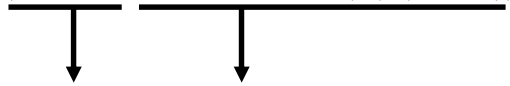
(LAMBDA (Lista-Lambda) (Cuerpo))

El primer elemento debe ser el símbolo Lambda. El segundo elemento debe ser una lista, denominada la lista lambda, y especifica los nombres de los parámetros de la función.

El cuerpo consiste en cero o más formas, que se evalúan secuencialmente, la función retorna el valor o valores que se obtengan de evaluar la ultima de las formas del cuerpo. Dado que dicha expresión no liga un nombre al procedimiento definido
Por ejemplo:

```
((LAMBDA (x) (+ x 3)) 5)                    → 8
((LAMBDA (a b) (+ a (* b 3))) 4 5)          → 19
((LAMBDA (M N) (+ M N)) 2 3)                → 5
((LAMBDA (X Y) (+ (* X X) (* Y Y))) 3 4)    → 25
```

```
(DEFUN CUADRADO (N) (* N N))
```



```
(LAMBDA (N) (* N N))
```

Cada elemento de una lista lambda es un parámetro o bien una palabra clave (estas comienzan con &).

Es incorrecto usar una expresión Lambda si la misma expresión aparece en distintos lugares para este caso es mejor definir un procedimiento con Defun en lugar de ponerlo repetidas veces con Lambda. Debido a que al definir el procedimiento en un solo lugar se ahorra espacio de almacenamiento, y si se desea modificarlo o reparar un error no se haría en varios lugares.

4.4 Funciones como Argumentos.

En Lisp una función es además un objeto de datos que puede ser suministrado a otra función como argumento. Esta posibilidad contribuye a la facilidad con que Lisp se puede adaptar a las necesidades de cualquier programa mediante la incorporación de nuevas funciones que en su comportamiento resultan idénticas a las primitivas.

Un programa que admite funciones como datos debe también suministrar alguna manera de invocarlas. Esto se logra mediante la funciones APPLY y FUNCALL.

4.4.1 APPLY

Aplica una función (que recibe como primer argumento) a una lista de argumentos. La función puede ser un objeto de código compilado, una expresión-lambda, o un símbolo. Siempre y cuando dicho símbolo no sea una forma especial. Una función Apply tiene la sintaxis siguiente:

(APPLY Función Lista-Args)

Por ejemplo:

(APPLY '+ '(2 4 6)) → 12
(APPLY #'(lambda (x y z)(+ x y z)) '(2 4 6)) → 12
(APPLY 'quote '(2 4 6)) → Error: Macro or special

4.4.2 FUNCALL

Aplica la función que se da como primer argumento al valor de los otros argumentos, es decir, permite definir procedimientos que tengan procedimientos como argumentos. Una función Funcall tiene la sintaxis siguiente:

(FUNCALL 'función Arg1 Arg2 ... ArgN)

Por ejemplo:

(FUNCALL '+ 1 2 3) → 6
(FUNCALL #'(lambda (x) (* 2 x)) 8) → 16
(FUNCALL 'append '(Verde Rojo) '(Azul Rosa)) → (VERDE ROJO AZUL ROSA)

Un ejemplo mas practico es el caso de declarar una función y luego esta llamarla con dichas funciones para lograr ver en que se diferencian.

(DEFUN HO (X Y) (+ X 5 (* Y 1))) →HO
(SETQ A 2) → 2
(FUNCALL 'HO 1 2) → 8
(FUNCALL 'HO 1 A) → 8
(APPLY 'HO '(1 2)) →8
(APPLY 'HO '(1 A)) → Error: In * of (A 1) arguments should be of type NUMBER.

La principal diferencia es que en FUNCALL hay que proporcionar tantos argumentos adicionales como argumentos necesita la función, y en APPLY todos los argumentos se pueden sintetizar en una lista.

4.4.3 FUNCTION

Esta función le indica al compilador que enlace y optimice un argumento tal como si fuera una función primitiva. FUNCTION es idéntico a la función QUOTE excepto en que fuerza la compilación del argumento de la misma manera que lo haría DEFUN.

Si incluimos las funciones internas en expresiones del tipo (function (lambda <parámetros> <expresiones> ...) en lugar de '(lambda <parámetros> <expresiones> ...), nos aseguraremos que el código sea enlazado (linked) y optimizado en tiempo de compilación en lugar de ser simplemente evaluado en tiempo de ejecución.

Las expresiones-LAMBDA compiladas contendrán información para su depuración al ser cargadas. Lo que generara un incremento de la velocidad de ejecución al crearse el correspondiente código optimizado en lenguaje máquina.

Por ejemplo: El compilador no puede optimizar la expresión lambda precedida de QUOTE en la siguiente expresión:
(MAPCAR '(LAMBDA (x) (* x x)) '(1 2 3)) →Error: Argument is not a function

Pero una vez incluida la expresión dentro de FUNCTION el compilador podrá optimizar la expresión lambda:
(MAPCAR (FUNCTION (LAMBDA (x) (* x x))) '(1 2 3)) → (1 4 9)

4.5 El Valor de los Parámetros

Los valores asociados a los parámetros al iniciar la ejecución de un procedimiento quedan aislados de otras asignaciones, como si Lisp levantara una cerca para evitar confusiones.
Por ejemplo siempre que el procedimiento Saludo se Lisp levanta una cerca alrededor del cuerpo para proteger cualquier valor que tenga el parámetro.

```
(DEFUN Saludo (Lis)
        (CONS 'Hola Lis))          → Saludo
```

Supóngase que Lis tiene un valor diferente del empleado en Saludo:

```
(SETF Lis '( Pedro Juarez))        →(PEDRO JUAREZ)
```

Y que continuación se llame al procedimiento pasándole directamente el valor:

```
(Saludo '(Juan Perez))             →(HOLA JUAN PEREZ)
```

En cuanto Saludo termina el valor de Lis aparece sin cambio alguno.

```
Lis                                → (PEDRO JUAREZ)
```

Así en este caso, el valor de Lis si se le hubiese asignado, y antes de que Saludo se utilice es el mismo después de que Lisp 4.3 termina de evaluar el cuerpo de Saludo.

En el caso de una variable que este en el cuerpo del procedimiento pero no es un parámetro no existe una cerca virtual que aislara todas las ligaduras del parámetro en el procedimiento. Por ejemplo:

```
(DEFUN Saludo ()
        (Setf Lis (Lista 'Hola Mundo)))   → Saludo
```

Saludo no posee parámetro de tal manera que Lis no lo es, causando que su valor no se aísle del valor que tiene fuera de la cerca construida al evaluarse Saludo.

```
(Saludo)                  → (HOLA MUNDO)
(Setf Lis '(Isla Bonita))  → (ISLA BONITA)
(Saludo)                  → (HOLA MUNDO)
Lis                       → (HOLA MUNDO)
```

De manera que dentro de un procedimiento existen dos tipos de variables, las que se encuentran aisladas por cercas virtuales y las que no lo están. Teniendo cada tipo de variable su nombre:

- **Variable Léxica:** Es aquella que se encuentra aislada por una cerca virtual, considerándose las que están dentro de un procedimiento del cual son parámetros.

- **Variable Especial:** Es aquella que no se encuentra aislada por una cerca virtual, considerándose las que están dentro de un procedimiento del cual no son parámetros.

El efecto secundario de Defun es establecer la definición de un procedimiento, donde efecto secundario es cuando se ha realizado un procedimiento y persiste después de devolver su valor.

A diferencia de muchas funciones primitivas, las funciones definidas por el usuario deben suministrar valores para todos los argumentos declarados. Defun no es una función normal, sino lo que se incluye entre las llamadas formas especiales, en el sentido de que sus argumentos no son evaluados. Serán evaluados al llamar a la nueva función, sustituyendo sus argumentos por los valores que se desea procesar.

Así se ha creado una nueva función Lisp que puede llamarse desde el nivel superior o anidada en otras funciones.

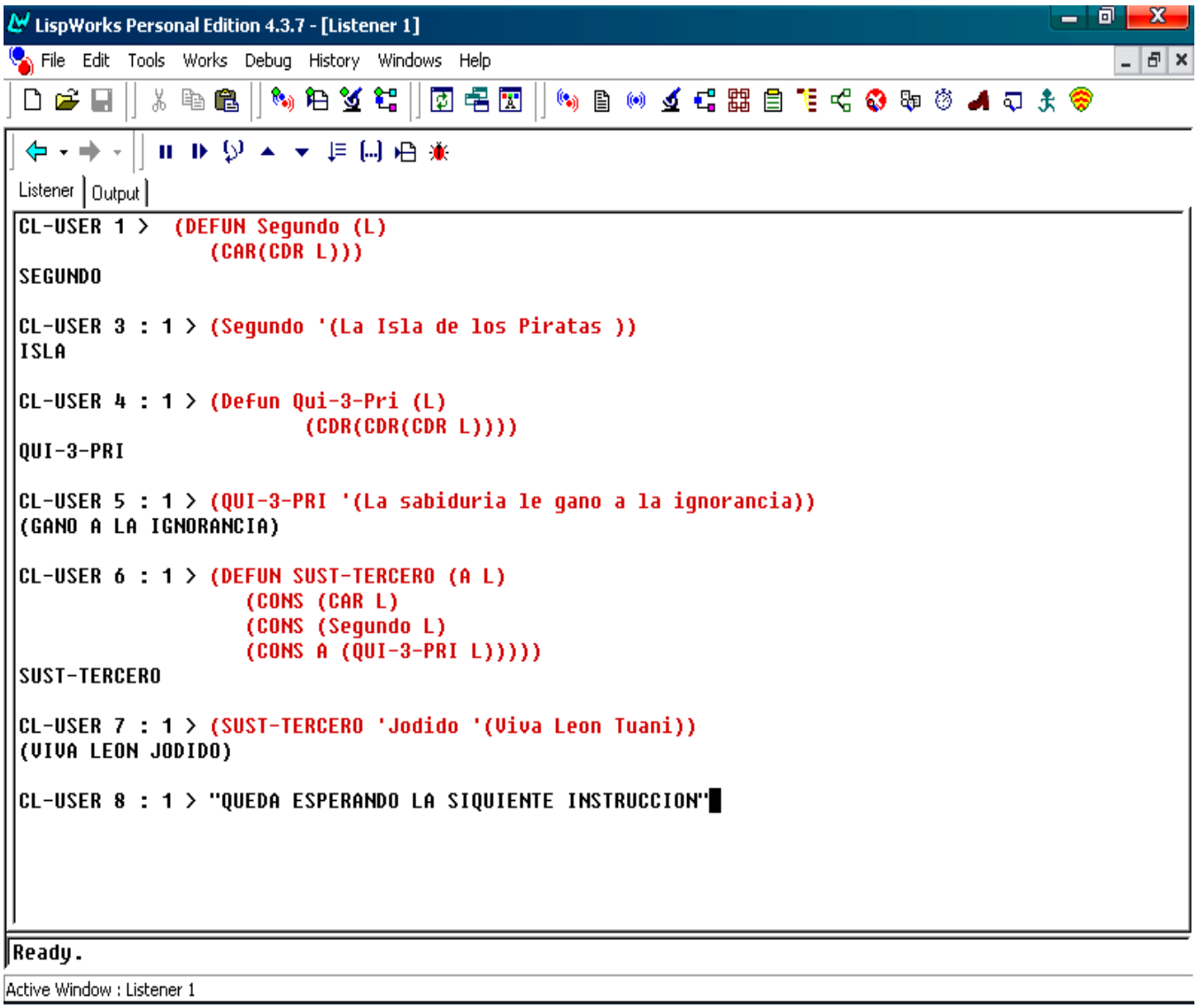
4.6 Ejemplos Prácticos:

La definición del procedimiento Segundo consiste en que se le pasa una lista y que devuelve el segundo elemento de esta, como la primitiva Second de Lisp.

El procedimiento Qui-3-Pri consiste en que se le pasa una lista y que devuelve el resto de esta sin los 3 primeros elementos como la primitiva Cdddr de Lisp.

Yel procedimiento SUST-TERCERO consiste en que L (lista) y a (átomo) que devuelve la lista formada sustituyendo el tercer elemento de L por a.

Solución:



Unidad

5

PREDICADOS Y ESTRUCTURAS DE CONTROL

A continuación se estudiaran las numerosas pruebas que se pueden realizar en Lisp 4.3 denominadas formalmente predicados. Estas pruebas combinadas con las estructuras de control permiten indicar al lenguaje como variar lo que esta sucediendo, conforme al avance de los procedimientos logrando hacer estos mas complicados y eficientes.

Contenido:

5 PREDICADOS

Un predicado es una función que chequea si sus argumentos cumplen una determinada condición y retorna NIL si no la cumplen o algo distinto de NIL, si efectivamente la cumplen. En otros lenguajes, estos predicados básicos se definen normalmente vía operadores "relacionales" y "booleanos".
Por convenio, los nombres de los predicados en Lisp suelen terminar en p.

5.1 Predicados Relacionados con Tipos de Datos

Lisp 4.3 proporciona una gran variedad de predicados predefinidos para comprobar el tipo de cualquier objeto. Los predicados no sólo se refieren a los tipos de datos de Lisp (números, átomos, listas, etc.), también identifican subconjuntos dentro de sus tipos (números positivos, el cero, etc.).

Predicado	Función	Ejemplo
Null	Retorna cierto si es una lista vacía, y falso en otro caso.	(null ’()) → T
Atom	Retorna cierto si no es una celda cons, y falso en otro caso.	(atom ’a) → T (atom 3/35) → T
Symbolp	Retorna cierto si es un símbolo y falso en otro caso.	(symbolp ‘a) → T (symbolp a) → NIL
Consp	Retorna cierto si el objeto es una celda cons, falso en otro caso.	(consp '(A B)) → T (consp ()) → NIL
Listp	Retorna cierto si es una celda cons o la lista vacía, falso en otro caso.	(listp ’(pontiac cadillac chevrolet)) T (listp 99) NIL
Numberp	Retorna cierto si es algún tipo de número, y falso en otro caso.	(numberp 6) →T (numberp ’6) →T (numberp 'A) → NIL

En el caso de Consp, hay que tener en cuenta que la lista vacía no es una celda cons. Listp no chequea si la lista es una lista propia o una lista impropia (terminada en un átomo distinto de NIL).

5.2 Predicados de Igualdad

Hay varios predicados que sirven para chequear si dos objetos son iguales: eq (el mas especıfico), eql, equal y equalp (el mas general). Si dos objetos satisfacen alguno de estos predicados, entonces también satisfacen aquellos que sean mas generales, es decir, si dos objetos satisfacen eql, también satisfacen equal y equalp.

1. Eq: Hay que destacar que los objetos que se representan externamente de la misma forma no son necesariamente iguales en el sentido de eq.

(EQ x y)

Es cierto si y solo si **x** e **y** son el mismo objeto, es decir, si ocupan la misma posición de memoria.
Por ejemplo:

(eq ’a ’b)	→Nil
(eq ’a ’a)	→T
(progn (setq x (cons ’a ’b)) (eq x x))	→T
(progn (setq x ’(a . b)) (eq x x))	→T
(eq "Foo" (copy=seq "Foo"))	→Nil
(eq "FOO" "foo")	→Nil

2. Eql: El predicado eql hace las mismas comprobaciones que eq excepto que si los argumentos son caracteres o números entonces se hace la comparación de sus valores. Sirve para averiguar si dos objetos son conceptualmente iguales mientras que eq indica si son iguales desde el punto de vista de la implementación.

(EQL x y)

Este predicado es cierto si sus argumentos son eq o si son números del mismo tipo y con el mismo valor o si son caracteres y su valor es el mismo.
Por ejemplo:

(eql ’a ’b)	→Nil
(eql ’a ’a)	→T
(eql 3 3)	→T
(eql 3 3.0)	→Nil

Para comparar los caracteres de dos cadenas se debe usar equal, equalp, string o string-equal.

3. Equal: Este predicado es cierto si sus argumentos son estructuralmente similares (isomorfos). Se podría decir que dos objetos son equal si su representación impresa es la misma.

(EQUAL x y)

La comparación para números es la misma que hace eql. La comparación entre símbolos es la misma que hace eq. Hay ciertos objetos compuestos que son equal si son del mismo tipo y sus componentes son equal.

Este chequeo se implementa de forma recursiva y puede no acabar para determinadas estructuras circulares. Dos conses son equal si su car es equal y su cdr es equal.

Por otra parte, dos arrays son equal si son eq, excepto para las cadenas de caracteres, que son comparados elemento a elemento. En las cadenas de caracteres se consideran distintas las mayúsculas de las minúsculas.
Por ejemplo:

(equal (cons ’a ’b) (cons ’a ’c))	→Nil
(equal ’(a . b) ’(a . b))	→T
(progn (setq x (cons ’a ’b)) (equal x x))	→T
(progn (setq x ’(a . b)) (equal x x))	→T
(equal #\A #\A)	→T
(equal "Foo" (copy=seq "Foo"))	→T
(equal "FOO" "foo")	→Nil

4. Equalp: Dos objetos son equalp si son equal. Si son caracteres y son los mismos, independientemente de que estén en mayúsculas o minúsculas, si son números y tienen el mismo valor numérico, independientemente de que sean de tipo diferente o si tienen componentes y cada una de ellas es equalp.

(EQUALP x y)

Por ejemplo:

(equalp 3.0 3.0)	→T
(equalp #c(3 =4.0) #c(3 =4))	→T
(equalp (cons ’a ’b) (cons ’a ’c))	→Nil
(equalp (cons ’a ’b) (cons ’a ’b))	→T
(progn (setq x (cons ’a ’b)) (equalp x x))	→T
(equalp "Foo" "Foo")	→T
(equalp "Foo" (copy=seq "Foo"))	→T

5.3 Predicados Aplicables a Números

Predicados	Función	Ejemplo
Zerop num	Retorna cierto si el número es cero, y falso en otro caso.	(zerop 0) T (zerop 1) Nil (zerop (/(+ 2 3)(* 3 8))) Nil
Plusp num	Retorna cierto si el número es mayor que cero y falso en otro caso.	(Plusp 1) T (Plusp 0) Nil (Plusp (- (+ 4 2) 7)) Nil
Minusp num	Retorna cierto si el numero es menor que cero y falso en otro caso.	(Minusp 1) Nil (Minusp 0) Nil (Minusp (- (+ 6 4) 11)) T
Oddp num-ent	Retorna cierto para un numero impar y falso en otro caso. El argumento debe ser un número entero.	(Oddp 8) Nil (Oddp 1) T
Evenp num-ent	Retorna cierto si el número es par y falso en otro caso. El numero debe se entero.	(Evenp (* 3 5 8)) T (Evenp 1) Nil

5.4 Predicados Aritméticos o Numéricos

1. (= n1 ... nN): Devuelve T si los valores de todos los argumentos son iguales, Nil, en caso contrario.
Por ejemplo:

(= 10 (+ 3 7)) → T
(= 2 2.0 (+ 1 1)) → T
(= 1 2 3) → NIL
(= 1 2 1) → NIL

2. (/= n1 ... nN): Devuelve T si los valores de todos los argumentos son distintos, Nil, en caso contrario.
Por ejemplo:

(/= 10 (+ 3 7)) → NIL
(/= 2 2.0 (+ 1 1)) → NIL
(/= 1 2 3) → T
(/= 1 2 1) → NIL

3. (> n1 ... nN): Devuelve T si n1 > ... > nN, Nil, en otro caso.
Por ejemplo:

(> 4 3 2 1) → T
(> 4 3 3 2) → NIL

4. (< n1 ... nN): Devuelve T si n1 < ... < nN, NIL, en otro caso.
Por ejemplo:

(< 1 2 3 4) → T
(< 1 3 3 4) → NIL

5. (>= n1 ... nN): Devuelve T si n1 >= ... >= nN, NIL, en otro caso.
Por ejemplo:

(>= 4 3 3 2) → T
(>= 4 3 3 5) → NIL

6. (<= n1 ... nN): Devuelve T si n1 <= ... <= nN, NIL, en otro caso.
Por ejemplo:

(<= 2 3 3 4) → T
(<= 5 3 3 4) → NIL

Estas funciones toman uno o más argumentos y retornan cierto si la secuencia de argumentos satisface la condición.

= todos iguales.
/= todos diferentes.
< Secuencia monótona creciente.
> Secuencia monótona decreciente.
<= secuencia monótona no decreciente.
>= secuencia monótona no creciente.

5.5 Estructuras de Control

Las funciones en Lisp pueden evaluarse en serie, condicional, iterativa o recursivamente.

Lisp tiene estructuras que permiten controlar el acceso a variables así como el flujo de ejecución de los programas. La construcción de programas en Lisp se basa en la aplicación de funciones, pudiendo estas ser recursivas.

5.6 Operadores lógicos

Para utilizar las formas de control condicionales es necesario escribir expresiones lógicas complejas. AND y OR se utilizan, respectivamente, para la conjunción y disyunción lógica de condiciones. Ambas admiten un número indeterminado de argumentos.

Hay tres operadores que actúan sobre valores booleanos, que son And, Or y Not son estructuras de control dado que sus argumentos se evalúan de una forma condicional (es decir, no siempre son evaluados).

- **And:** Evalúa una a una las formas que se pasan como argumento y si alguna de ellas devuelve NIL, se detiene y devuelve NIL. Si todas devuelven valores distintos de NIL, devuelve el valor de la última forma.

  (And expr1……..exprN)

  Por ejemplo:

  (AND 1 2 3) → 3
  (AND 1 NIL 3) → NIL

- **Or:** Evalúa una a una las formas y si alguna devuelve un valor distinto de NIL, se detiene y devuelve ese valor. En caso contrario continúa con las siguientes formas o devuelve NIL si no queda ninguna.

  (Or expr1……..exprN)

  Por ejemplo:

  (OR 22 5 nil)→ 22
  (OR (EQ 'AER 'WILL) (EQUAL 'ALL 'WILL)) → NIL
  (OR NIL nil nil 150) → 150
  (OR NIL 2 3) →2

- **Not:** Retorna T si expresión es NIL y retorna NIL en cualquier otro caso.

  (Not expresión)

  Por ejemplo:

  (NOT (< (+ 10 10) 22)) → NIL
  (NOT (= (+ 11 15) 30)) → T
  (NOT Nil) → T

5.7 Constantes y Variables

Dado que algunos objetos en Lisp se usan para representar programas, dentro de un programa no se puede hacer referencia a un dato simplemente escribiendo su representación, puesto que sería ambiguo, si se desea utilizar el valor escrito o el resultado de su evaluación. La forma especial quote sirve para resolver esta ambigüedad.

5.7.1 Definición de Constantes

Para definir constantes en Lisp se utiliza **Defconstant**.
Por ejemplo:

(DEFCONSTANT Clases '(Clases Español Matemática Ingles)) →Clases

La función Defconstant devuelve siempre el nombre de la constante que se acaba de definir. El valor de una constante no se puede modificar. Según el ejemplo, Clases es una constante cuyo valor es una lista con el nombre completo de las clases. Es decir, siempre que se evalúe Clases, el resultado será su valor:

Clases	→	(Clases Español Matemática Ingles)
(REVERSE Clases)	→	(Ingles Matemática Español Clases)

Obsérvese que el símbolo Clases aparece dos veces en la definición, como símbolo dentro de la lista y como nombre de la constante. En realidad, tanto las constantes como las variables no son sino simples símbolos a los que se ha asociado un valor.

5.7.2 Definición de Variables Globales

Para definir variables globales se utiliza **Defvar**. Existe el convenio de definir las variables globales con dos asteriscos para que las expresiones que las utilizan sean más legibles.
Por ejemplo:

(DEFVAR *Tarzan* '(Rey de la selva))	→* Tarzan *
(DEFVAR *Lazarillo*)	→*Lazarillo*

Defvar permite definir variables globales dándoles un valor inicial (opcional), devuelve el nombre de la variable que se acaba de definir. Si se define una misma variable más de una vez, sólo tiene efecto la primera definición.

Por ejemplo, si se define de nuevo la variable * Hercules * con otro valor inicial, permanece el valor dado la primera vez:

(DEFVAR * Hercules * '(Es hijo de Zeus))	→* Hercules *
* Hercules *	→ (Es hijo de Zeus)

Si se define una variable, pero no se le da valor, utilizar esa variable dará error porque no tiene un valor asignado.
Por ejemplo, evaluar *LAZARILLO* da error porque es una variable que no tiene asociado ningún valor:

```
*LAZARILLO*
*** - EVAL: la variable *LAZARILLO* no tiene ningún valor
```

5.7.3 Definición de Variables Locales

Let y Let*: Las estructuras LET y LET* sirven para la definición de variables locales e incluyen un cuerpo en donde tienen validez las mismas. La sintaxis es idéntica para ambas:

```
(LET ((var1 valor1)
      (var2 valor2)
      ...
      (varN valorN) )
     cuerpo)
```

En un Let todos los valores iniciales de las variables se asignan en paralelo, lo que significa que una variable no puede depender de otra definida en el mismo LET, mientras que en un LET* el valor de las variables se asigna secuencialmente, y por tanto puede depender de las variables definidas anteriormente.

Por ejemplo:
(Setf Lista-Comida '(desayuno almuerzo cena)) → (DESAYUNO ALMUERZO CENA)

```
(LET ((desa (FIRST Lista-Comida))          ; Relaciona desa con un valor inicial
      (cen (LAST Lista-Comida)))           ; Relaciona cen con un valor inicial
      (CONS desa cen))                     ; Combina desa y cen
```

→(DESAYUNO CENA)

A continuación un ejemplo de Let* y su equivalente Let:

```
(SETF x 'RELOJ)
(LET* (( x 'AGUJA)
       (y x))
       (list x y))
→(AGUJA  AGUJA)
```

```
(SETF x 'RELOJ)
(LET (( x 'AGUJA))
       (Let (y x))
       (list x y)))
→(AGUJA  AGUJA)
```

5.8 Condicionales de Control

Los predicados se utilizan para determinar de entre varias formas cual debe evaluarse, con frecuencia la elección se determina mediante predicados de conjunción If.

5.8.1 El símbolo **IF** va seguido de una forma de prueba, así como de algo para evaluar y devolver cuando la evaluación de la prueba es diferente de Nil.
La estructura **IF** tiene la forma:

(IF <condición> <forma-then> <forma-else>)

Por ejemplo:

(Setf Día 'Lunes)	→Lunes
(IF (SYMBOLP Día) 'Día es un símbolo ' Día no es símbolo)	→Día es un símbolo
(Setf Día '1)	→1
(IF (SYMBOLP Día) 'Día es un símbolo ' Día no es símbolo)	→Día no es un símbolo
(IF (= (SETQ A 3) 4) 1 0)	→0
(IF (= A 3) 1 0)	→1
(IF (= A 4) (+ A 2))	→Nil

Las formas If forman un pequeño grupo cuyos miembros son tratados como casos especiales por los intérpretes y compiladores de Lisp.

5.8.2 Las formas **When** y **Unless** son variaciones de la forma If.

En especial se puede usar When en lugar de IF siempre y cuando el resultado de la condición sea T, When equivale a un If sin parte Else. Su estructura tiene la forma:

(When <condición> <forma a evaluar si es T>)

De manera similar se usa Unless en lugar de IF siempre y cuando el resultado de la condición sea Nil, Unless equivale a un If sin parte Then. Su estructura tiene la forma:

(Unless <condición> <forma a evaluar si es Nil>)

Por ejemplo:

```
(WHEN (= (+ 2 3) 5) (SETQ A 1) (SETQ B 5) (+ A B))          → 6
(WHEN (= (+ 2 3) 6) (SETQ A 1) (SETQ B 5) (+ A B))          → NIL
(UNLESS ( = ( + 2 3) 5) (SETQ A 1) (SETQ B 5) (+ A B))      → NIL
(UNLESS ( = ( + 2 3) 6) (SETQ A 1) (SETQ B 5) (+ A B))      → 6

(SETQ X ? )                                                 → ?
(WHEN (EQL 3 X) 'Hacer esto si X es 3)
(UNLESS (EQL 3 X) 'Si no es 3 hacer esto)
```

Si X tiene valor 3 el resultado del When será "Hacer esto si X es 3", y el del Unless Nil. Si no vale 3, el resultado del When será Nil y el del Unless será "Si no es 3 hacer esto".

5.8.3 La estructura **Cond** evalúa las condiciones hasta que encuentra una que se cumpla, y entonces evalúa la expresión asociada a ésta. Si ninguna es verdadera, devuelve Nil. Su estructura tiene la forma:

(Cond (<condición 1> <expresión 1-1> …)
(<condición 2> <expresión 2-1>…)
.
.
.
(<condición n> <expresión n-1>…)

Normalmente se utiliza como última condición T para realizar una acción en el caso de que ninguna otra condición se cumpla. Si alguna condición no tiene ninguna expresión asociada se devuelve el valor de la condición.
Por ejemplo:

```
(DEFUN Nota (N)
     (COND ((< N 60) 'Reprobado)
           ((< N 70) 'Aprobado)
           ((< N 90) 'Excelente)
           ((<= N 10) 'Sobresaliente)
            (T 'ERROR)))                    → Nota
(Nota 80)                                   →Excelente


(SETF Borracho ‘(Caballito Perla RonPlata))   → (Caballito Perla RonPlata)
(COND ((> (LENGTH Borracho) 4) ‘ Metanol)
          (Borracho ‘ AmanecedeGoma)
           (T ‘ NoTomo)                     → AmanecedeGoma
```

5.8.4 La estructura **Case** evalúa una expresión y compara el resultado con los posibles valores sin evaluar, y si encuentra alguno coincidente evalúa la forma asociada.
Su estructura tiene la forma:

(case expresión-clave
(clave-1 expresión-valor 1-1 expresión-valor 1-2 ...)
(clave-2 expresión-valor 2-1 ...)
.
.
.
(clave n-1 expresión-valor n-1 ...))

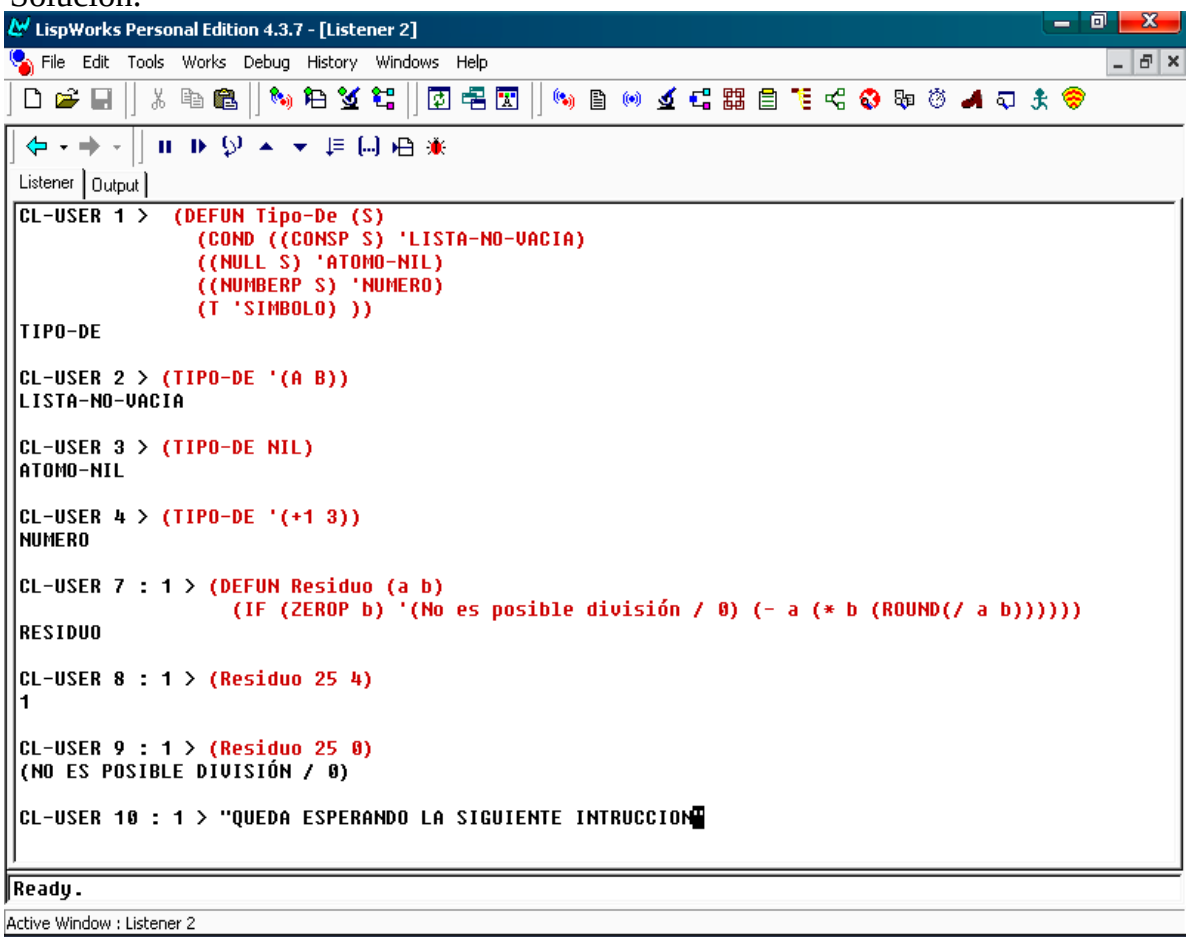
Es decir, se evalúa expresión-clave, dando un valor, luego se inspeccionan secuencialmente las claves y si se encuentra alguna clave-n que coincide con el valor de expresión-clave, se evalúan secuencialmente la correspondiente expresión-valor, y se devuelve el valor de la última de ellas. Si ninguna clave y ningún elemento de una lista-clave cumplen alguna de estas condiciones, se devuelve NIL.

Si la clave-n última es T u OTHERWISE, se evalúan secuencialmente la correspondiente expresión-valor, y se devuelve el valor de la última de ellas. En el caso de que los valores tengan la forma de listas, Case usa como criterio la función Member.
Por ejemplo:

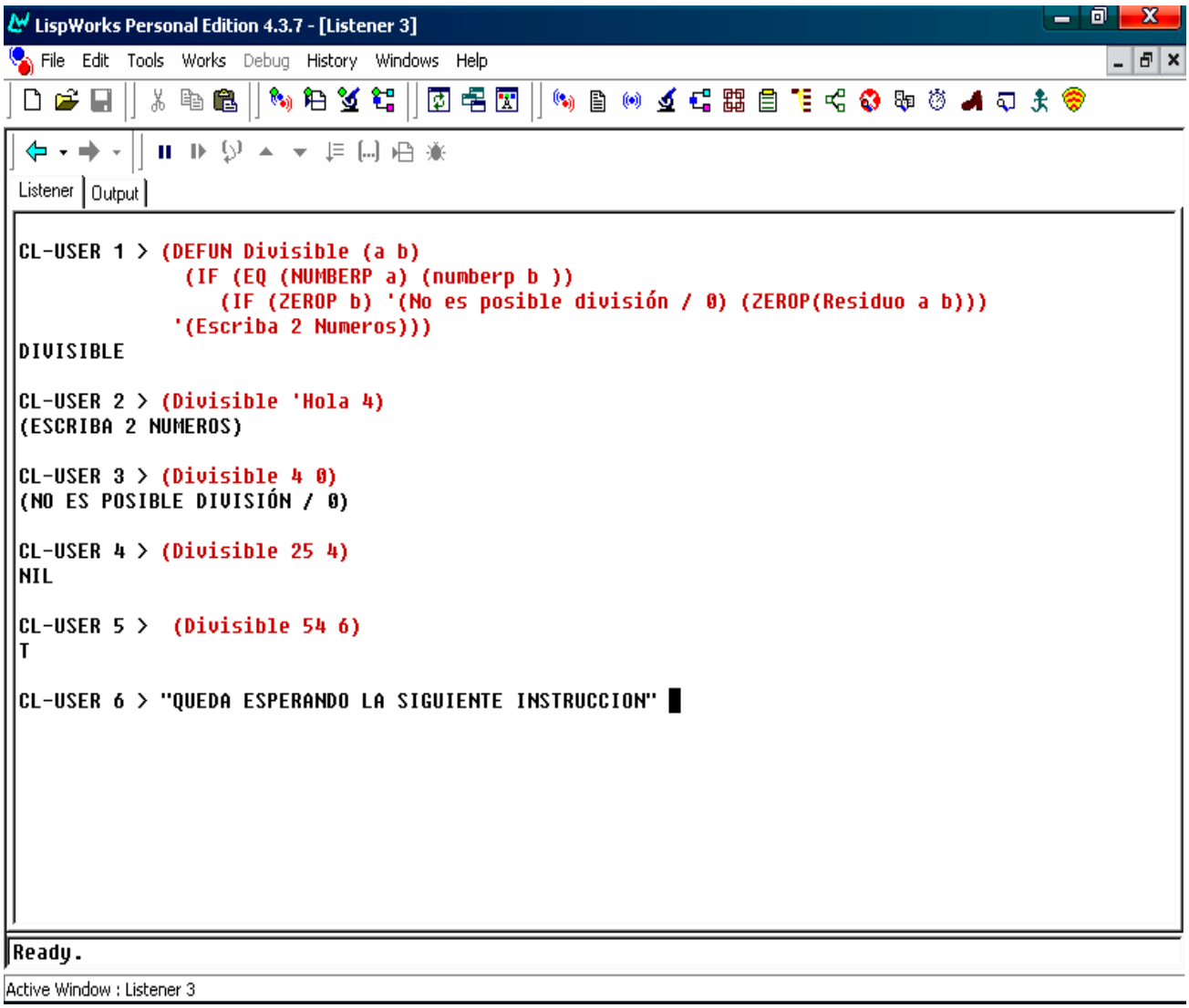
```
(DEFUN Meses (N)
      (CASE N
          (1 'Enero)
          (2 'Febrero)
          (3 'Marzo)
          (4 'Otros )))                    → Meses
(Meses 3)                                  → Marzo
(SETF  Casado ‘Lunamiel)                   → Lunamiel
(CASE Casado
        (Lunamiel ‘Hawai)
         (Divorciado ‘BuscanPareja)
         (Viuda ‘Quedoconreales))          → Hawai
```

5.9 Ejemplos Prácticos

La definición del procedimiento Tipo-De consiste en que se le pasa un argumento cualquiera y devuelve como valor su tipo.
El procedimiento Residuo consiste en dividir dos argumentos cualquiera devolver el resto de esta división.
Solución:



La definición del procedimiento Divisible consiste en dividir dos argumentos cualquiera devolver si el primero es divisible del segundo, observe que ocupa la función Residuo.
Solución:



Unidad

6

ITERACION Y RECURSION

Una característica de Lisp es su gran facilidad para programar funcional y recursivamente. En esta unidad se introduce la iteración desde los ejemplos más sencillos, para aprender de forma sólida los aspectos básicos.

Así como la abstracción de procedimientos que es un aspecto fundamental que no hay que descuidar a la hora de abordar un problema complejo. La recursión e iteración de procedimientos son unas de las varias estrategia generales para controlar el desarrollo de los cálculos considerándose estos una clase de estrategia de control.

Contenido:

6 ITERACIÓN

Aunque la recursividad es la forma primaria de expresar los procesos repetitivos en Lisp, en algunas situaciones se prefiere la especificación de bucles "iterativos".
A continuación se presentan diversas construcciones que permiten realizar iteraciones.

6.1 Iteración General

Las funciones especiales que permiten definir iteraciones son Do y Do*:

```
(DO ( (var1 valorinic1 funcionpaso1)
      (var2 valorinic2 funcionpaso2)
      .
      .
      .
      (varN valorinicN funcionpasoN) )
      ( condición-final cuerpo-final )
cuerpo)
```

Algunas de las partes de Do pueden no aparecer o estar vacías. El funcionamiento de Do se puede explicar de la siguiente manera:

1) Se inicializan las variables.

2) Mientras no sea cierta la condición-final hacer:
- Evaluar cuerpo (puede ser una o varias sentencias, es igual a un Progn implícito).
- Modificar las variables según la función paso.

3) Se evalúa cuerpo-final (Progn implícito). El resultado de éste es el resultado del Do.
Por ejemplo:

```
(DO ((A 3 (- A 1)))                    →3
    ((= A 0) 'FIN)                     →2
    (PRINT A) )                        →1
                                       →Fin
( DEFUN Exponencial (m n )
  ( DO (( Resultado 1)
        (exp  n))
        (( ZEROP exp ) Resultado)
        (SETF Resultado ( *  m  Resultado))
        (SETF exp  (- exp 1))))                → Exponencial
 (Exponencial 4 2)                             →16
```

DO* es similar a Do. Do hace las asignaciones iniciales y las actualizaciones de los valores en paralelo como Let, mientras que DO* las hace secuencialmente como Let*.
Por ejemplo:

```
(DO* ((A 2 (- A 1))
      (B A A) )                    →2
      ((= B 0) 'FIN)               →1
       (PRINT B) )                 → Fin
```

6.2 Iteración Indefinida

La construcción Loop es la más simple. No controla ninguna variable y simplemente ejecuta el cuerpo del bucle repetidamente. Evalúa sucesivamente s1,..., sN hasta encontrar un Return.

(LOOP s1...sN)

Se evalúa cada forma de izquierda a derecha. Cuando se ha evaluado la ultima forma entonces se comienza de nuevo evaluando la primera, y así sucesivamente, en un ciclo infinito. Esta construcción debe abandonarse explícitamente usando, Return.
Por ejemplo:

```
(PROGN (SETQ S '(A B))
   (LOOP (IF S (PRINT (POP S))
      (RETURN 'FIN) )))
→A
→B
→ FIN
```

6.3 Construcciones para Iteración Simple

A continuación se presentan las construcciones Dolist y Dotimes.

6.3.1 Dolist

(DOLIST (var L resultado) s1...sN)
Es lo mismo que:

```
(DO* ((LISTA-AUX L (CDR L))
       (VAR (CAR LISTA-AUX) (CAR LISTA-AUX)) )
      ((NULL LISTA-AUX) RESULTADO)
S1 ... SN)
```

Permite iterar sobre los elementos de una lista. Es decir; asigna a var el primer elemento de la lista L, evalúa s1,..., sN, si L no tiene mas elementos, devuelve resultado, en otro caso, le asigna a var el siguiente elemento de L e itera el proceso.

Por ejemplo:

```
(DOLIST (Contador '(A B C) 'Fin)
   (PRINT Contador) )
→A
→B
→C
→Fin
```

6.3.2 Dotimes

(DOTIMES (var m resultado) s1...sN)
Es lo mismo que:
(DO ((VAR 0 (1+ VAR)))
((= VAR M) RESULTADO)
S1 ... SN)

Permite iterar sobre una secuencia de número enteros. Es decir, asigna a var el valor 0, evalúa s1,..., sN, aumenta el valor de var en 1, si dicho valor es igual a M devuelve resultado, en otro caso, itera el proceso.

Si el valor lımite es cero o negativo no se ejecuta ninguna iteración. Si se omite la forma resultado, el bucle retorna NIL.
Por ejemplo:

```
(DOTIMES (Contador 3 'fin)
   (PRINT Contador) )
→0
→1
→2
→FIN
```

Un ejemplo de uso de Dotimes para procesar cadenas de caracteres es la función que retorna cierto si la cadena de caracteres que se le pasa como argumento es la misma si se lee de izquierda a derecha y de derecha a izquierda. Por ejemplo:

```
(DEFUN Ver-DI-ID (string)
    (LET ((start 0)  (END (LENGTH string)))
       (DOTIMES (k (FLOOR (- end start) 2) t)
          (UNLESS (CHAR-EQUAL  (char string (+ start k))
                       (char string (- end k 1)))
             (RETURN Nil)))))
```

6.4 El grupo Progn, Go, Return

6.4.1 La Función Progn

(PROGN L s1...sN)

L es una lista de variables locales que toman el valor inicial Nil. PROGN evalúa secuencialmente s1,..., sN, si no se detiene con un Return, devuelve el valor de sN.
Por ejemplo:

(PROGN (SETF a ‘x)(SETF b ‘y)(SETF c ‘z)) →Z

Cuando, dentro de un Progn, se encuentra un símbolo, este no es evaluado, sino que se toma como etiqueta.

6.4.2 La Función Return

(RETURN s)

Da error si no está dentro de un PROG o un LOOP, cuando lo está, devuelve el valor de s, y detiene el ciclo.

6.4.3 La Función GO

(GO simb)

Transfiere el control a la etiqueta simb del PROG. Da error si se usa fuera de un PROG.
Por ejemplo:

```
(PROG (L)
   ETIQUETA
   (COND ((= (LENGTH L) 3) (RETURN L))
            (T (SETQ L (CONS 7 L))
               (GO ETIQUETA) )))
→ (7 7 7)
```

6.5 RECURSIVIDAD

La definición de una función es recursiva si en el cuerpo de la definición aparece la llamada a la función.

La recursividad consiste en resolver un problema dividiéndolo en un paso sencillo y un problema más pequeño. Cuando el problema llega a ser suficientemente pequeño tiene una solución trivial.

Veamos, por ejemplo, la definición recursiva del factorial de un número natural es:

$$\text{factorial}(n) = \begin{cases} 1 & n=0 \\ n * \text{factorial}(n-1) & n>0 \end{cases}$$

El caso n=0 se llama caso trivial porque tiene solución directa, y el caso n>0 se llama caso general porque se soluciona recursivamente.
Por ejemplo:

¿Cómo calcularíamos recursivamente el factorial de 3?

Factorial (3)= 3* factorial (2) ; caso general
Factorial (2)= 2* factorial (1) ; caso general
Factorial (1)= 1* factorial (0) ; caso general
Factorial (0)= 1 ; caso trivial

Un planteamiento recursivo puede tener uno o más casos triviales y uno o más casos generales. Para que la recursión funcione bien debe cumplir:

1) Que el problema se haga más pequeño en cada llamada recursiva y la disminución lleve necesariamente a un caso trivial. Por ejemplo: en el factorial, n se disminuye en uno en cada llamada recursiva, con lo que necesariamente en n llamadas se alcanzará n=0.

2) Que en cada caso se devuelva el resultado adecuado, tanto en los casos triviales como en los generales. Por ejemplo: si para n=0 devolviéramos cero, el factorial de cualquier número saldría, erróneamente, cero.

3) Que se hayan considerado todos los casos posibles, tanto triviales como generales.

Por ejemplo:

```
(DEFUN Factorial (N)
    (IF (ZEROP N) 1 (* N (Factorial (1- N)))))   → Factorial
(Factorial 4)                                    → 24

(DEFUN Cont (Lista)
   (IF (NULL Lista) 0   (+ 1 (Cont (REST Lista))))) → CONT

(Setf Tony '(Blanco Alto Recio))          → (BLANCO ALTO RECIO)
(Cont Tony)                               → 3
```

Las versiones recursivas sobre listas son fáciles de ver, debido a la definición recursiva de las mismas. Una lista es o bien vacía o bien un elemento concatenado a una lista.

Aunque la recursividad es el dispositivo primario para definir funciones en Lisp, en algunas ocasiones es necesaria la iteración. Además, la mayoría de las funciones requieren el uso de variables locales para disponer de un almacenamiento temporal mientras realizan sus tareas.

Cualquiera de estos dos requerimientos fuerza al uso de la característica progn dentro de la definición de una función.

6.6 Recursión Doble

Un procedimiento posee recursión doble cuando puede llamarse a si mismo dos veces y no solo una. Se puede utilizar para analizar expresiones anidadas.

Un ejemplo sencillo de la recursión doble es una función que se le pasa una lista como argumento, donde dicho argumento son expresiones Lisp.
A la función la llamaremos Con-Atom ya que esta cuenta átomos en una expresión dada su definición es:

```
(DEFUN Con-Atom (L)
    (COND ((NULL L ) 0)
          ((ATOM L) 1)
          (T ( + (Con-Atom (FIRST L))  (Con-Atom(REST L))))))

→ CON-ATOM
Con-Atom '(setf Mirno '(List Machete Zambumba (cons Morena Chela)))
→ 9
```

En cada llamada, el argumento de Con-Atom se divide en dos partes pequeñas, una vez que se tiene la respuesta para ambas partes se suman los resultados y se devuelve el valor.

Se han abordado procedimientos con recursión sencilla como Cont y Factorial, así como con recursión doble, Con-Atom. Generalmente los procedimientos con recursión sencilla son adecuados para trabajar con varios tipos de secuencias tales como explorar elementos de una lista.

Los procedimientos con recursión doble son adecuados cuando se van a explorar árboles, como todas las partes de una expresión, incluso las que están profundamente anidadas.

6.7 Recursión Terminal

Cada vez que un problema se convierte en otro nuevo y no son necesarios cálculos adicionales para su resolución se dice que el nuevo problema es una reducción del problema original.
Cuando se defina un procedimiento recursivo de manera que todas sus llamadas recursivas a si mismo son reducciones, este procedimiento es recursivo terminal.

Por ejemplo el procedimiento anterior Con-Atom que cuenta átomos lo definiremos para contar los elementos de una lista:

```
(DEFUN Con-Lista (L)
        (COND ((NULL L ) 0)
                ((ATOM L) "No es una lista")
                ( T (+ 1    (Con-Lista(REST L))))))
```

Cada vez que Con-Lista opera en una lista que no sea vacía, se llama a si mismo con un problema mas sencillo, la segunda llamada no es una reducción de la primera, ocasionando que el resultado de la segunda llamada deba ser incrementado en uno antes de que se terminen los cálculos, obteniendo su valor real hasta que ha pasado por las llamadas intermedias.

Pero si definimos el procedimiento de la manera siguiente:

```
(DEFUN Con-Lista (L)
        (Con-Lista-Best L 0))

(DEFUN Con-Lista-Best (L Res)
        (COND ((NULL L ) Res)
                ((ATOM L) "No es una lista")
                ( T ( Con-Lista-Best  (REST L)  (+ 1 Res)  ))))
```

Con-Lista-Best es un procedimiento recursivo terminal porque cada problema que se le entrega se convierte en otro problema para si mismo, de modo que la respuesta al nuevo problema es la respuesta al problema original, obteniendo así el resultado ignorando todas las llamadas intermedias.

Aún cuando no se llamen directamente a si mismo son recursivos por que cada vez que ocurran dos llamadas sucesivas a un procedimiento la segunda de ellas una reducción de la primera.

6.8 Abstracción de Procedimientos

La atracción de procedimientos es un concepto eficaz ya que permite realizar programas grandes en virtud de los siguientes beneficios determinantes:

- Ayuda a pensar en un nivel superior, al permitir obviar detalles de cómo se hacen las cosas en un nivel inferior. Puede programar de arriba abajo trabajando primero en los procedimientos de nivel superior y posponiendo los de nivel inferior.

- Permite mantener definiciones de procedimientos breves y comprensibles.

Por ejemplo una función llamada Apellidos que devuelve todos los apellidos de una lista:

```
(SETF Nombres '((Sir Antonio Lopez) (Señor Pepe Galvez)
(Don Pepe Cuena)  (Mr Carlos Sierra)))

(DEFUN Apellidos (Lis)
  (IF (NULL Lis) Nil
      (LIST*  (FIRST(REVERSE (CAR Lis))) (Apellidos (CDR Lis)))))
```

Para realizar este procedimiento se deben de hacer tres cosas:

1- Confirmar si la lista esta vacía.
2- Extraer el primer apellido.
3- Extraer el primer elemento de la lista y devolver el resto de esta.
4- Combinar los apellidos extraídos.

En nuestro procedimiento se realizaron estos 4 procesos ejecutando de manera directas las funciones de Lisp. Pero si utilizáramos abstracción de procedimientos tendríamos 3 procedimientos auxiliares:

```
(DEFUN Apellidos (Lis)
     (IF (NULL Lis) Nil
         (Comb-Apell  (Sacar-Apell Lis ) (Apellidos (Cort-Apell Lis)))))
```

Al definir Apellidos de este modo, no importa como se haría la combinación de apellidos y extracción de elementos que realizan los procesos auxiliares siempre y cuando produzcan el mismo resultado.

```
(DEFUN  Sacar-Apell (Lis)
        (FIRST(REVERSE (CAR Lis))))

(DEFUN  Cort-Apell(Lis)
        (REST Lis))

(DEFUN  Comb-Apell (ap apn)
        (LIST* ap apn))
```

Formando así un nivel con un solo procedimiento, los procedimientos auxiliares son de nivel inferior y cualquier procedimiento que se use Apellidos forma parte de un nivel superior. Los procedimientos auxiliares tienen nombre largos que llegan hacer comentario lo que contribuye ser más clara la definición.

Es mas recomendable usar nombres largos para los procedimientos puesto que así se explican por sí mismo en cualquier lugar que se empleen. Este uso se traduce así en una especie de comentarios automáticos.

Siempre que los procedimiento se trabajan en niveles, se hace atracción de procedimiento en cuyo caso los niveles se denominan niveles de atracción esto se separan unos de otros por barreras de atracción las cuales sirven para aislar los detalles de implementación de los procedimientos de cada nivel.

6.9 Análisis de la Recursión

Para llevar a cabo el análisis de un procedimiento recursivo se utiliza una primitiva de Lisp la cual es Trace que permite averiguar por qué una función no se comporta como se esperaba, quizá debido al valor de sus parámetros.

Imprimiendo información cuando se inicia y termina una llamada a dicho procedimiento, se imprime el nombre del procedimiento, argumentos cuando se llama y el resultado cuando termina. Tiene la sintaxis siguiente:

(TRACE Fun1 … FunN)

Por ejemplo:

```
(TRACE Con-Lista)        → (CON-LISTA)

(Con-Lista '(La sabiduria es buena))
```

```
0 CON-LISTA > ...
  >> L : (LA SABIDURIA ES BUENA)
  1 CON-LISTA > ...
    >> L : (SABIDURIA ES BUENA)
    2 CON-LISTA > ...
      >> L : (ES BUENA)
      3 CON-LISTA > ...
        >> L : (BUENA)
        4 CON-LISTA > ...
          >> L : NIL
        4 CON-LISTA < ...
          << VALUE-0 : 0
      3 CON-LISTA < ...
        << VALUE-0 : 1
    2 CON-LISTA < ...
      << VALUE-0 : 2
  1 CON-LISTA < ...
    << VALUE-0 : 3
0 CON-LISTA < ...
  << VALUE-0 : 4
4
```

Para llegar al resultado definitivo, debemos esperar a lo que devuelve cada uno de los ciclos de la recursión. Es evidente que resulta una pérdida de tiempo el esperar a que cada nivel devuelva ese mismo resultado hasta llegar al nivel superior.

Los compiladores Lisp más avanzados reconocerán una función de este tipo y cortarán el proceso en el instante que se obtiene el resultado del último nivel, mejorando sustancialmente la velocidad de operación de las funciones.

La primitiva Untrace deshabilita la función Trace para los argumentos dados.

La recursión, aunque aporta soluciones de gran elegancia, debe ser utilizada con cautela, lo ideal sería usarla con funciones que no utilicen, o utilicen muy pocos argumentos, ya que la recursión coloca cada vez una nueva copia de la función en la memoria de pila junto con sus argumentos.

Es muy posible que una función efectúe tantas recursiones que se agote la memoria disponible provocando un error de desbordamiento de la pila.

Existen tres factores a considerar para saber cuando usar iteración o recursión:

1- Las funciones iterativas son usualmente más rápidas que sus contrapartes recursivas. Si la velocidad es importante, normalmente usaríamos la iteración.

2- Si la memoria de pila es un limitante, se preferirá la iteración sobre la recursión.

3- Algunos procedimientos se programan de manera recursiva de forma muy natural, y resultan prácticamente inabordables iterativamente. Aquí la elección es clara.

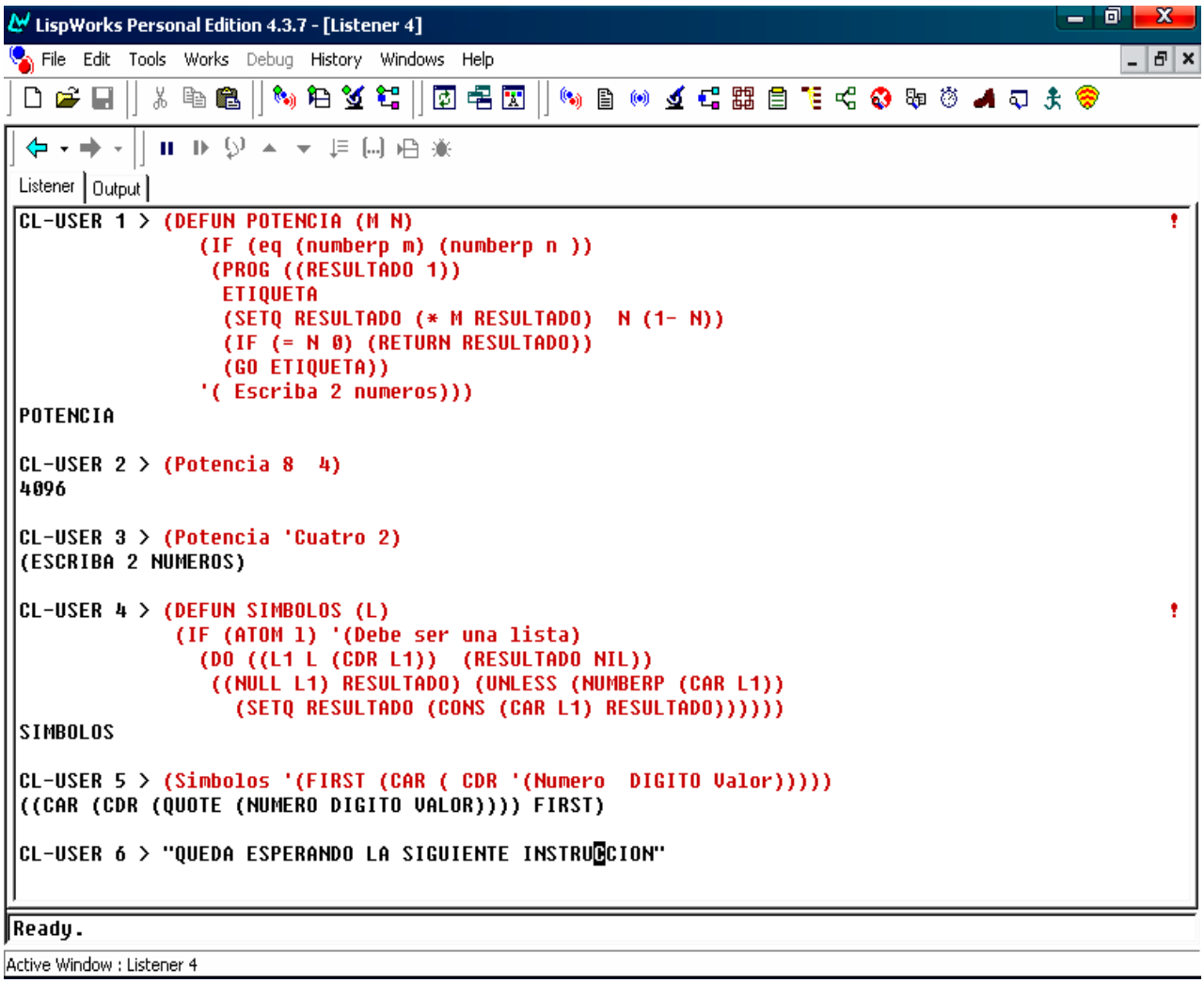
En Lisp muchas tareas implican el buscar a través de estructuras anidadas. Por ejemplo, las representaciones en árbol de los movimientos en un juego se representan mejor como listas anidadas.

Examinar este árbol implica un rastreo recursivo a través del mismo. Para este tipo de aplicación las funciones recursivas resultan una herramienta eficazmente esencial.

6.10 Ejemplos Prácticos

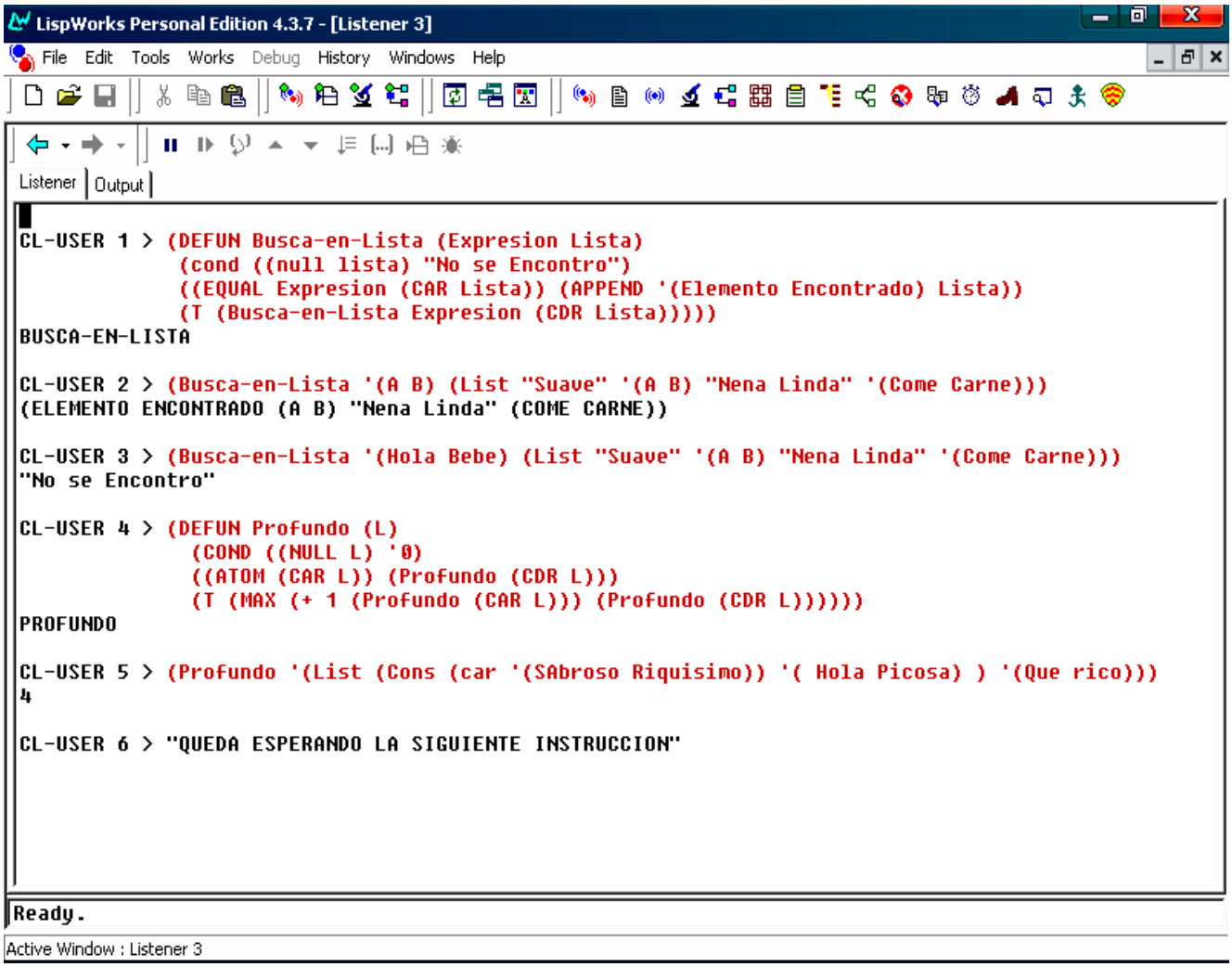
La definición del procedimiento Potencia consiste en que se le pasen dos argumentos numéricos cualquiera y devuelva el primero elevado al segundo de manera iterativa.

El procedimiento Simbolo consiste en que se le pase una lista como argumento y devuelve una lista con las funciones primitivas de Lisp de manera iterativa.
Solución:



El procedimiento Busca-en-Lista recibe como argumentos una expresión cualquiera y una lista. En caso que la expresión esté contenida en la lista, el procedimiento devolverá la expresión y todos los términos de la lista que ocurren después de la expresión. Los resultados obtenidos son los mismos que los de la función primitiva Member.

La definición del procedimiento Profundo consiste en analizar el máximo nivel de anidamiento de paréntesis en la lista que se pasa como argumento. Supondremos que la profundidad de NIL es 0. Empleando únicamente CAR, CDR, CONS y APPEND como funciones para manejo de listas.
Solución:



Unidad

7

LECTURA / ESCRITURA Y FICHEROS

Para concluir, esta unidad estudia las operaciones de entrada/salida, que permiten al usuario interactuar con el programa. La entrada en Lisp 4.3 es muy fácil ya que el lenguaje provee al usuario de primitivas que se utilizan para leer y devolver expresiones Lisp.

Así mismo, se explicaran primitivas para la creación y lectura de ficheros. Siendo este un método eficaz para leer expresiones o información de archivos, conforme se requieran para procesar.

Contenido:

7 LECTURA Y ESCRITURA

Lisp es un lenguaje interactivo, por lo que las funciones de entrada-salida se realizan principalmente sobre el terminal. La mayoría de las implementaciones permiten también el almacenamiento de archivos en memoria secundaria, pero esto es muy dependiente de la implementación.

7.1 Funciones de Lectura

7.1.1 Read: La función "Read" no tiene argumentos y hace que se introduzca una lista deteniendo la ejecución del programa hasta que se le introduce un dato.
Tiene la siguiente forma:

(READ)

Cuando se encuentra esta función, el programa espera que el usuario introduzca una lista, la cual se convertirá en el valor devuelto por esta función. Para asignar ese valor a una variable del programa, read puede combinarse dentro de una función **"setq",** como sigue: (setq X (read)).

En efecto, esto dice "asignar a X el siguiente valor de entrada". Por tanto, si escribimos “Jerónimos” en respuesta a la función read, Jerónimos será el valor de X.
Por ejemplo:

(SETQ L (READ))
Jerónimos → Tecleo
L → Jerónimos

(READ)
(CONS 'Hola '(Mundo Fresco)) → Tecleo
→ (CONS (QUOTE HOLA) (QUOTE (MUNDO FRESCO)))

Hemos dicho que Read lee una a una las expresiones Lisp. Pero en realidad no sólo lee, sino que lleva a cabo un análisis lexicográfico y sintáctico de lo leído, para determinar, por ejemplo, dónde acaba cada expresión, pero no la evalúa.

7.1.2 Eval: Si queremos evaluar el valor de una expresión devuelta por Read se usa eval, comando que calcula primero su argumento y después su resultado.
Tiene la siguiente forma:

(EVAL S)

Cuando Read es pasado como argumento al comando Eval, Read se evalúa porque es un argumento en un comando, luego hay una segunda evaluación ya que es el comando Eval.

Por ejemplo:

(EVAL (READ))
(CONS ‘Hola ‘(Mundo Fresco)) →Tecleo
→ (HOLA MUNDO)

7.1.3 Read-Line: Lee una línea del canal de entrada y devuelve dicho objeto. Tiene la siguiente forma: **(READ-LINE).**

Es decir, lee todos los caracteres hasta encontrar un salto de línea. La función devuelve la cadena formada por todos los caracteres de la línea (salvo el de salto). Read-Line devuelve también un segundo valor: "verdadero", si la línea acaba con el carácter de fin de fichero, "falso" en otro caso.

7.1.4 Read-Char: Lee un carácter del canal de entrada y devuelve dicho objeto. Tiene la siguiente forma: **(READ-CHAR).**

7.2 Variables de Escritura

7.2.1 Print-Base: Indica la base en que se escriben los números. Por defecto es 10. Puede variar de 2 a 36.
Por ejemplo:

PRINT-BASE → 10
14 → 14
(SETQ *PRINT-BASE* 2) → 10
14 →1110
(SETQ *PRINT-BASE* 10) → 10
16 → 16

7.2.2 Print-Length: Indica el número de elementos que se escriben.
Por ejemplo:

(SETQ *PRINT-LENGTH* 5) → 5
'(1 2 3 4 5 6 7 8 9) → (1 2 3 4 5...)

7.2.3 Print-Level: Indica el número de niveles de paréntesis que se escriben.
Por ejemplo:

(SETQ *PRINT-LEVEL* 2) → 2
'(Hola Mundo (Fresco Cool (Vida Rica)) (Beer Guaro(Goma Fea))
→ (HOLA MUNDO (FRESCO COOL #) (BEER GUARO #))

7.3 Funciones de Escritura

7.3.1 Print: La forma más directa de visualizar la salida sobre la pantalla es usar la función Print, la cual tiene la siguiente forma:

(PRINT s)

Aquí puede ser cualquier expresión en Lisp, y su valor se presentará sobre la pantalla como resultado de la ejecución de esta función.
Por ejemplo:

(PRINT (+ 7 3))

→ 10
→ 10

7.3.2 Prin1: Es como Print, excepto que no comienza en una nueva línea, escribiendo el valor de s, usando caracteres de control, la cual tiene la siguiente forma:

(PRIN1 s)

Por ejemplo:

(PRIN1 "A")
→ "A"
→ "A"

7.3.3 Princ: Escribe el valor de s, sin usar caracteres de control, tiene la siguiente forma:

(PRINC S)

Se utiliza para suprimir las barras verticales, las cuales se utilizan para encerrar los átomos que contienen caracteres especiales (como "$" y "espacio en blanco", los cuales no se permiten normalmente dentro de un átomo).

Por ejemplo, si quisiéramos que un átomo tuviera el valor ¡HURRA! Tendríamos que encerrarlo dentro de barras verticales, como las siguientes: | ¡HURRA!|. Si visualizáramos esto usando print o prin1, Las barras verticales también aparecerían.

PRINT '|¡ HURRA !| → ¡\ HURRA\ !
→ ¡\ HURRA\ !

PRIN1 '|¡ HURRA !| → ¡\ HURRA\ !
→ ¡\ HURRA\ !

princ '|¡ HURRA !| → ¡ HURRA !

7.3.4 Terpri: Escribe una línea en blanco.

(TERPRI)

7.3.5 Format: Para conseguir salidas formateadas se emplea la función Format, la cual tiene la siguiente forma:

(FORMAT Destino CAD ARG1 ... ARGN)

Permite realizar impresiones más elegantes, si destino es T significa la salida de datos estándar (la pantalla), retornando Nil.
Cad es una cadena de control y Arg1 …Argn es una sucesión de expresiones.

Si destino es Nil, Format devuelve la cadena descrita más adelante. En otro caso, Format se evalúa a NIL.
Concretamente la cadena de control se escribe literalmente en la corriente de salida, a excepción de los caracteres precedidos por ~, que se denominan directivas de formateo, cad puede contener caracteres de control.

Alguno de ellos son los siguientes:

- ~ % nueva línea.
- ~ D si el argumento es un numero decimal.
- ~ A si el argumento es un carácter ASCII.
- ~ B si el argumento es un número binario.
- ~ O si el argumento es un numero octal.
- ~ X si el argumento es un número hexadecimal.
- ~ S Imprimir según las reglas de evaluación.

Por ejemplo:
(FORMAT T “~%LINEA 1 ~%LINEA 2”)
→ LINEA 1
→ LINEA 2
→ NIL

(FORMAT Nil "~%LINEA 1 ~%LINEA 2")
→ "
→ LINEA 1
→ LINEA 2"

(FORMAT T "~%EL CUADRADO DE ~D ES ~D" 3 (* 3 3))
→ EL CUADRADO DE 3 ES 9
→ NIL

(FORMAT T "~%10 EN BINARIO ES ~B, EN OCTAL ES ~O Y EN HEXADECIMALES ~X" 10 10 10)
→10 EN BINARIO ES 1010, EN OCTAL ES 12 Y EN HEXADECIMAL ES A
→NIL

7.4 FICHEROS

Cuando se abre un fichero se construye un objeto de tipo stream, que sirve como intermediario entre el sistema de ficheros y el entorno Lisp. Las operaciones que se hagan sobre el flujo de datos (stream) se reflejan en acciones sobre el fichero. La asociación entre el fichero y el flujo de datos termina al cerrar el fichero.

7.5 Apertura de un Fichero

7.5.1 Open: Retorna un flujo conectado al fichero que se especifica como primer argumento, la cual tiene la siguiente forma:

Nom-Can (OPEN NOMBRE-FICHERO :DIRECTION :CLAVE :IF-EXISTS :IF-DOES-NOT-EXIST :EXTERNAL-FORMAT))

Ese nombre se puede dar como cadena de caracteres, objeto de tipo pathname o objeto de tipo stream (lo habitual es que se utilice una cadena de caracteres).

Las palabras clave tienen el siguiente significado.

- Nom-Can: es el nombre asignado al canal que va a estar unido al fichero que se esta abriendo.
- Nombre-Fichero: es el camino hacia el fichero, incluyendo su nombre.
- Direction: es un modificador para indicar que el canal que se va a construir es de entrada, salida o ambas cosas. El valor de Clave puede ser:
  - Input: flujo de entrada.
  - Output: flujo de salida.
  - Io: flujo de entrada/salida.

Esto es útil si se desea comprobar si un fichero existe sin establecer realmente la conexión con dicho fichero para leer o escribir en el.

- If-exists: Este argumento especifica la acción que se va a llevar a cabo si la dirección del flujo es de salida o de entrada/salida. Si la dirección es de entrada, este argumento no se tiene en cuenta. Su valor puede ser:
  - Error: Se produce un error.
  - Overwrite: Sobrescribe el fichero.
  - Append: Se añade información al fichero.
  - Supersede: Equivale a eliminar el fichero existente y crear uno nuevo.

- If-does-not-exist: Especifica la acción que se llevara a cabo si el fichero especificado no existe. Su valor puede ser:
  - Error: Se produce un error.
  - Create: Se crea un fichero vacío.

Por ejemplo:
(SETQ Fic (OPEN "Fich.DAT" :DIRECTION :OUTPUT))
→#<STREAM::LATIN-1-FILE-STREAM C:\Documents and Settings\ Snatch\ Fich.DAT >

(FORMAT FIC "HolaMundo") → NIL
(CLOSE Fic) → T

(SETQ Leer (OPEN "Fich.DAT" :DIRECTION :Input))
→#<STREAM::LATIN-1-FILE-STREAM C:\Documents and Settings\Snatch\Fich.DAT>

(PRINT (READ Leer)) → HOLAMUNDO
→ HOLAMUNDO

7.5.2 With-Open-File

La manera más sencilla y recomendable de crear corrientes para comunicarse con ficheros, y además ligarlas a símbolos, es emplear la forma:
(WITH-OPEN-FILE (Nom-Can Nombre-Fichero :DIRECTION: CLAVE) S1...SN

Entre otras razones por el implícito **Unwind-Protect** que lleva consigo, que no es más que el manejo de errores. Es decir, el archivo siempre queda cerrado, aun cuando la evaluación de With-Open-File haya terminado prematuramente asegurando así que el fichero no se perjudique, siendo el sistema el que garantiza que el fichero quede cerrado.

Las palabras clave tienen el siguiente significado.

- Nom-Can: símbolo el cual se le asocia el nombre del canal, evaluando las expresiones S1,..., SN y devuelve el valor de SN.
- Nombre-Fichero: nombre del fichero.
- Si la CLAVE es: INPUT, las expresiones SI pueden ser de lectura como: (READ SIMB).
- Si la CLAVE es: OUTPUT, las expresiones SI pueden ser de escritura como: (FORMAT SIMB S), (PRINT S SIMB), (PRINC S SIMB) o (TERPRI SIMB).

Por ejemplo:

```
(WITH-OPEN-FILE (Fic "Fichero.DAT" :DIRECTION :OUTPUT)
                (FORMAT Fic "Rich ~%Will ~%Axl") )
                → NIL
(WITH-OPEN-FILE (Leer "Fichero.DAT" :DIRECTION :INPUT)
    (PRINT (READ Leer))
    (PRINT (READ Leer))
    (PRINT (READ Leer))
    'FIN )
                                → Rich
                                → Will
                                → All
                                → Fin
```

7.6 Cierre de un Fichero

Close: La función que cierra el canal nombrado por SIMB es close.

(CLOSE SIMB)

Se puede utilizar cuando un flujo esta asociado a un fichero, cerrando dicho flujo e impidiendo, por tanto, volver a leer o escribir en dicho fichero (a menos que se vuelva a abrir y a asociar a un stream).

Por ejemplo:

```
(CLOSE Fic)        → T
```

7.7 Cargar un Archivo

La Función Load: Carga el fichero Nombre.lsp

(LOAD Nombre)

Cargar un fichero en el entorno Lisp significa leer el fichero evaluando cada una de las formas que aparecen en dicho fichero. Los programas se almacenan en ficheros y estos son susceptibles de ser cargados con load.

De este modo, podemos escribir el programa mediante cualquier editor de texto y evaluarlo en el intérprete de diversas maneras.

Para finalizar la sesión, se puede hacer mediante las funciones (quit) o (exit).

7.8 Funciones Sobre Ficheros

1. Dribble: Escribe lo que sale en pantalla en el fichero de nombre s y también cierra el fichero abierto.

(DRIBBLE s)

Por ejemplo:

(DRIBBLE Fic) → CL-USER
→ CL-USER
(DRIBBLE) → ; Closed dribble to #<STREAM::LATIN-1-FILE-STREAM C:\Documents and Settings\Snatch\Fich.DAT>

3. Rename-File: Se cambia el nombre al fichero. El primer argumento y el segundo suelen ser cadenas de caracteres.

(RENAME-FILE Fichero Nom-Nuevo)

Por ejemplo:

(RENAME-FILE "Fich.dat" "Hola")
→ #P"Hola.dat"
→ #P"C:/Documents and Settings/Snatch/Fich.dat"
→ #P"C:/Documents and Settings/Snatch/Hola.dat"

4. Probe-File: Retorna falso si no existe ningún fichero cuyo nombre coincida con el argumento que se le pasa.

(PROBE-FILE Fichero)

Por ejemplo:

(PROBE-FILE "Hola.dat") → #P"C:/Documents and Settings/Snatch/Hola.dat "

5. Delete-File: Se borra el fichero especificado. El argumento suele ser una cadena de caracteres.

(DELETE-FILE Fichero)

Por ejemplo:

(DELETE-FILE "Hola.dat") → T

7.9 Ejemplos prácticos

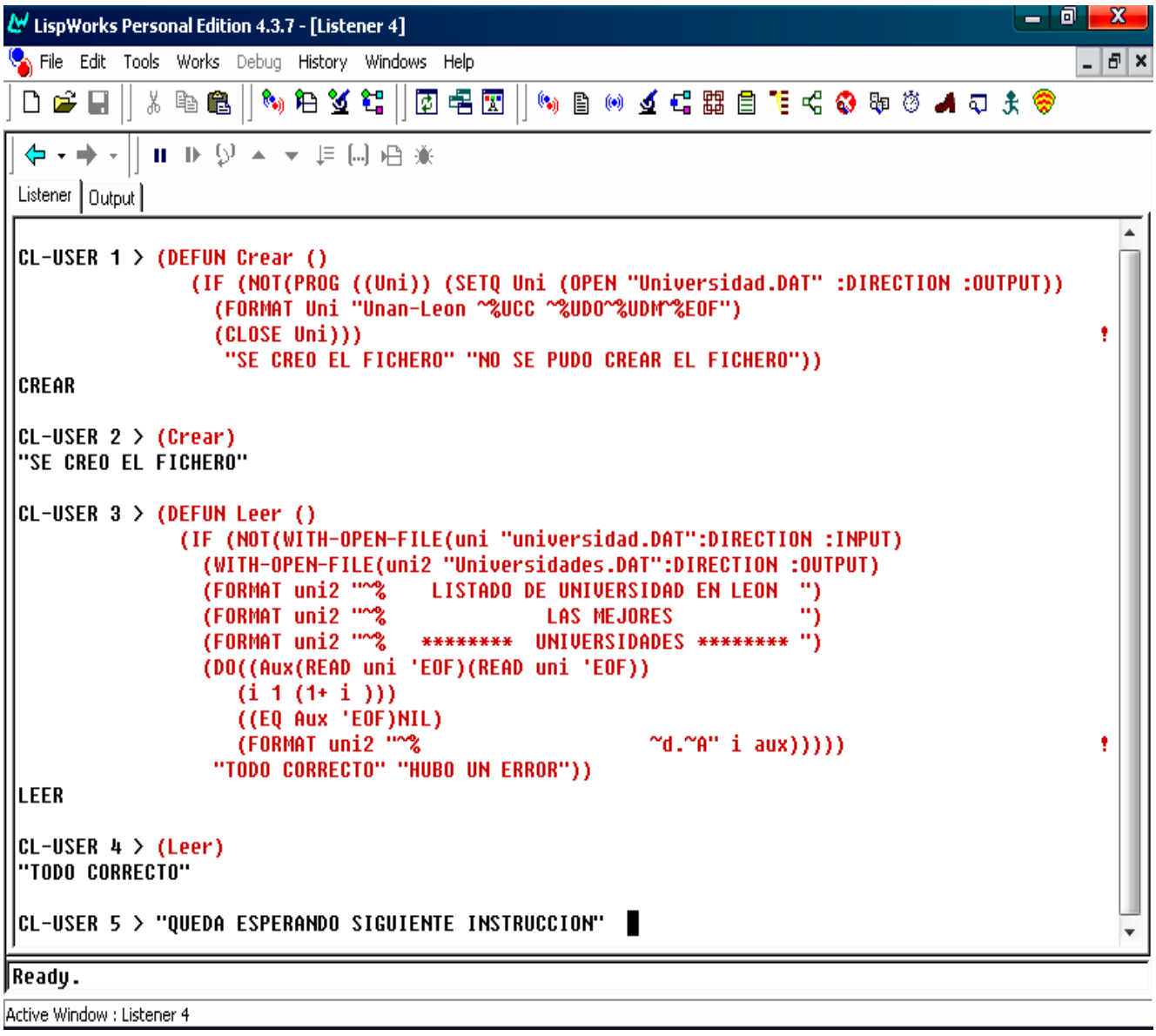
La definición del procedimiento Crear consiste en crear un fichero y guardar la información siguiente:

```
UNAN-LEÓN
UCC
UDO
UDM
```

La definición del procedimiento Leer consiste en abrir el fichero creado con anterioridad y guardar en otro fichero la información siguiente:

LISTADO DE UNIVERSIDAD EN LEÓN LAS MEJORES
********** UNIVERSIDADES **********
1. UNAN-LEÓN.
2. UCC.
3. UDO.
4. UDM.

Para comprobar mas específicamente de la creación de dichos archivos se abre la ruta C:\Documents and Settings\Snatch donde Snatch es el actual usuario de la Pc.



PRACTICAS

PRACTICAS PROPUESTAS

A medida que se avanza en la asignatura en aspectos teóricos, es también de mucha importancia ir realizando ejercicios para que el estudiante vaya observando el vínculo que existe entre lo teórico y lo práctico. Las prácticas a continuación le servirán al alumno para reforzar la visión que se tiene de las cuestiones teóricas provenientes tanto de libros como de otros documentos.

En este apartado se reúnen una serie de ejercicios para el aprendizaje del lenguaje Lisp. Además, se incluirá una práctica dirigida especialmente para el intérprete que se utilizó en el transcurso del trabajo*: Xanalys LispWorks Personal Edition Version 4.3*.

Cada práctica esta dividida en grupos de ejercicios referentes a distintos temas y también pertenecientes a distintos niveles de dificultad los cuales irán creciendo progresivamente. Se sugiere que el alumno realice todos los ejercicios para alcanzar un mejor conocimiento del lenguaje.

Esta sección de prácticas propuestas es para la asignatura de Inteligencia Artificial y Sistemas Expertos que es impartida en el 5to. Año de la carrera de Ingeniería en Sistemas de Información en la UNAN-LEON.

Objetivos de las Prácticas Propuestas.

♣ Familiarizar al estudiante en el manejo del entorno integrado de desarrollo que ofrece el interprete *LispWorks* para la gestión de programas.

♣ Iniciar al alumno en la programación básica en un lenguaje meramente funcional como lo es *Lisp.*

♣ Que el alumno ponga en practica los conocimientos adquiridos en el transcurso de esta asignatura.

PRACTICA 1

ENTORNO DE DESARROLLO XANALYS LISPWORKS

Existen muchos intérpretes del lenguaje Lisp los cuales tienen solo pequeñas diferencias en la interfaz de cada uno, pero en la mayoría de ellos se pueden poner en práctica los conocimientos teóricos del lenguaje Lisp.

Esta primera sección de prácticas es una introducción al entorno de desarrollo del programa con el cual se trabajó durante todo el desarrollo de este soporte.

Se le mostrará al estudiante de la manera más sencilla como se manipula el intérprete Lisp: *Xanalys LispWorks Personal Edition Version 4.3.*

No existe intérprete alguno en versión español para el manejo de Lisp y a la vez todos los manuales del programa que se pueden conseguir por Internet vienen en idioma ingles.

Se le enseñará al estudiante las utilidades básicas necesarias para poder lidiar con el intérprete Xanalys LispWorks para que lleve a cabo su estudio del lenguaje y la posterior realización de las prácticas que están propuestas a continuación de esta.

En síntesis, esta práctica contendrá:

♣ Breve introducción del intérprete y su manera de trabajar.

♣ Las utilidades más necesarias del entorno de LispWorks.

♣ Forma de escribir, compilar y ejecutar las instrucciones Lisp.

Bibliografía.

*Internet:
www.xanalys.com

ENTORNO DE DESARROLLO XANALYS LISPWORKS.

El intérprete.

El intérprete es un programa que acepta expresiones, las evalúa y escribe el resultado de dicha evaluación. A este bucle de funcionamiento se le llama bucle de lee-evalúa-escribe (*read-eval-print loop*, en inglés). Todas las definiciones que se evalúen son recordadas por el intérprete mientras dure la sesión.

Cuando durante la evaluación se encuentra un error, el intérprete entra en un bucle de ruptura (*break loop*, en inglés). Se llama así, porque se rompe el funcionamiento normal del intérprete. Para cancelar el error y volver al bucle de funcionamiento normal basta con abortar el bucle de ruptura.

A continuación se introduce el intérprete que se utilizó durante todo el soporte y las sesiones de prácticas.

El intérprete LispWorks.

LispWorks es un intérprete y compilador de Lisp de distribución comercial. Para elaborar este soporte y la solución de las prácticas propuestas se utilizó LispWorks para Windows. El LispWorks ofrece un entorno de desarrollo integrado para la realización de aplicaciones en el sistema Ms-Dos. Este entorno permite cargar, interpretar, compilar y depurar código escrito en Lisp.

Conociendo el entorno.

LispWork es un programa bastante flexible ya que se puede instalar en Windows 98, Windows ME, Windows 2000 o también en Windows XP.

Se procederá a la instalación del paquete de LispWorks. Para ello se tecleara dos veces en el siguiente archivo ejecutable:

Una vez en la instalación, se recomienda aceptar todas las condiciones que se presenten.
Aceptar y Finalizar.

Ahora para ejecutar el programa:
1- Clic en Inicio de la barra de tareas.
2- Elegir Programas->Xanalys LispWorks 4.3->Personal Edition->LispWorks.

Después de una corta pausa (ya que el programa no es pesado) deberías ver una pantalla en la cual solo tienes que teclear OK. Luego se mostrará una ventana tal como lo muestra la Figura 1. Además, la ventana del Listener también aparecerá si tu imagen esta configurada para comenzar.

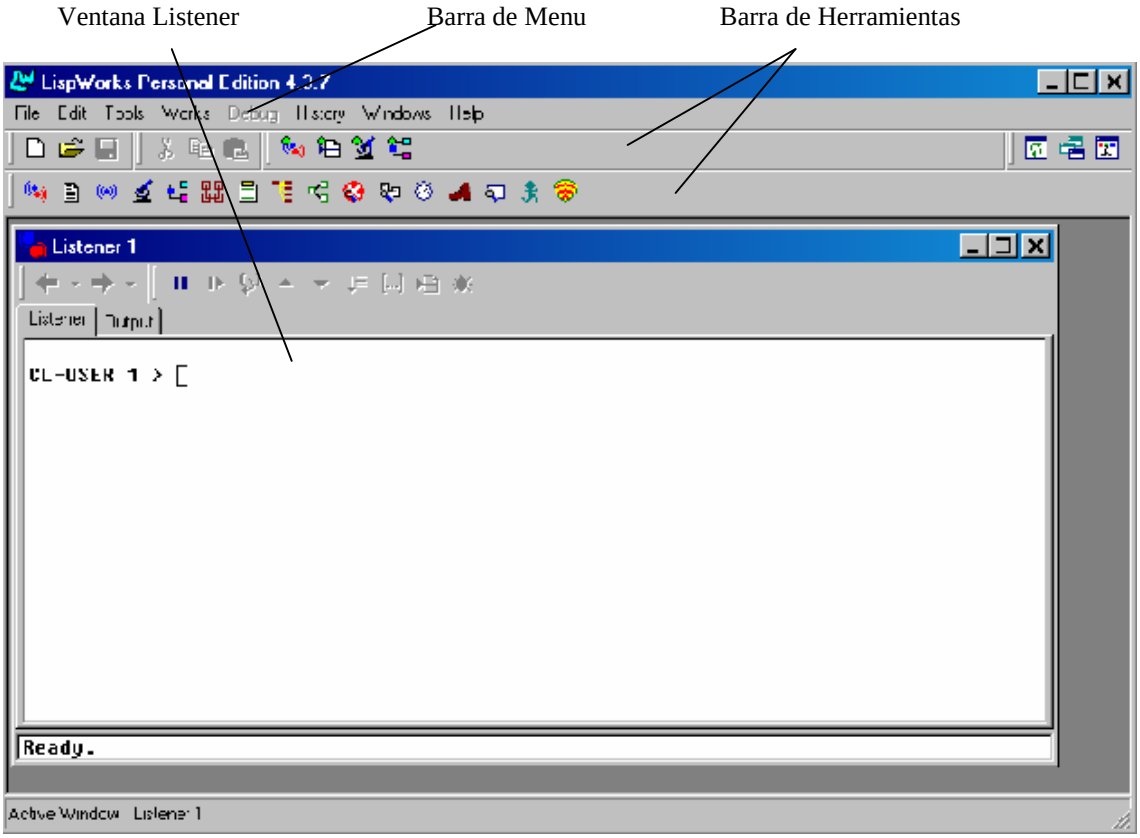


Figura 1

La Figura 1 será automáticamente mostrada siempre que comiences el entorno de LispWorks. Su barra de menú da acceso a varios comandos así como todas las herramientas del entorno. Su barra de herramientas da acceso rápido a algunos de los comandos más convenientes del programa. Pero en esta práctica solo se enfocaran los comandos que se usarán a lo largo del estudio.

Como en muchas otras aplicaciones, la barra de menú contiene *File*, *Edit*, *Tools*, *Windows* y *Helps* menús y un menú específico de LispWorks llamado *Works*.

El menú *File* te permite abrir o imprimir un archivo en un editor sin importar cual ventana esta activa. Cuando el Editor o el Listener esta activo, el menú *File* contiene otros comandos para diferentes operaciones sobre el archivo mostrado. El menú *Tools* te da acceso a todas las herramientas de Lisp. El menú *Windows* lista todas las ventanas LispWorks que estas usando.

Creando en Listener.

La herramienta Listener evalúa interactivamente todas las formas Lisp que escribas. Durante una típica sesión, tú evalúas trozos de código en el Listener, luego examinas los efectos en otras herramientas y al final regresas al Listener siempre que quieras evaluar otro trozo de código.
Un Listener es creado cuando comienzas el entorno Lisp. Si no tienes un Listener (chequea el menú Windows), empieza uno eligiendo Tools->Listener o presiona con el ratón en la barra de herramientas. Una ventana del Listener esta mostrada en la Figura 2.

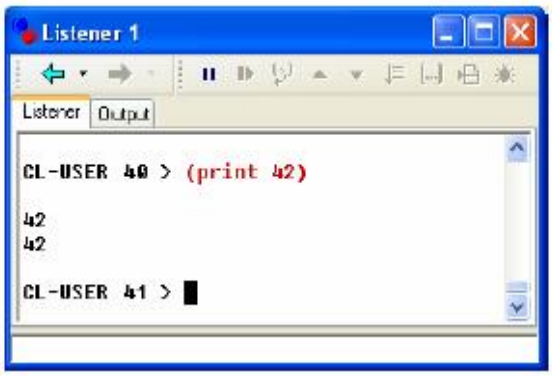


Figura 2

El Listener contiene dos pestañas: la de Listener y la de Output. Puedes intercambiar entre las dos pestañas al hacer clic. En el centro del Listener esta el área donde se llevaran a cabo todos los trabajos de escritura de programas y su compilación.

La interacción con el programa se realiza por medio del prompt, que en este caso es:

CL- USER 1 >

Aquí, se inicia el bucle lee-evalúa-escribe y el intérprete queda a la espera de qué expresiones simbólicas leer. Puedes evaluar las formas Lisp que desees en la vista Listener escribiendo la forma, seguido por un Return (Enter). Cualquier expresión que es escrita es mostrada en la interfaz de Listener.

Cuando escribimos una expresión simbólica, el intérprete no hace distinción entre mayúsculas o minúsculas.

```
CL- USER 1 >   (Cons  'A (CONS  'E (CoNs  'R (cons  'V  Nil ) ) ) )
(A E R V)
```

Añadir espacios en blanco o tabuladores no influye en la evaluación de lo que se escribe. Hasta que no se ha terminado de escribir la sentencia por completo, el intérprete no comienza a evaluarla.

```
CL- USER 2 >  (+ 2  3  4  5  6
                 4  2  1  0  3)
30
```

Cuando se produce un error en un programa, se produce una interrupción (break). El intérprete indica cual es y el nivel de bucle de ruptura. LispWorks saca un mensaje de error, entra en el depurador y presenta al usuario una o más posibles acciones de recomenzar.

Por ejemplo, si llamamos a la función * con un símbolo alfabético en lugar de con un número, se genera el siguiente error mostrado en la Figura 3:

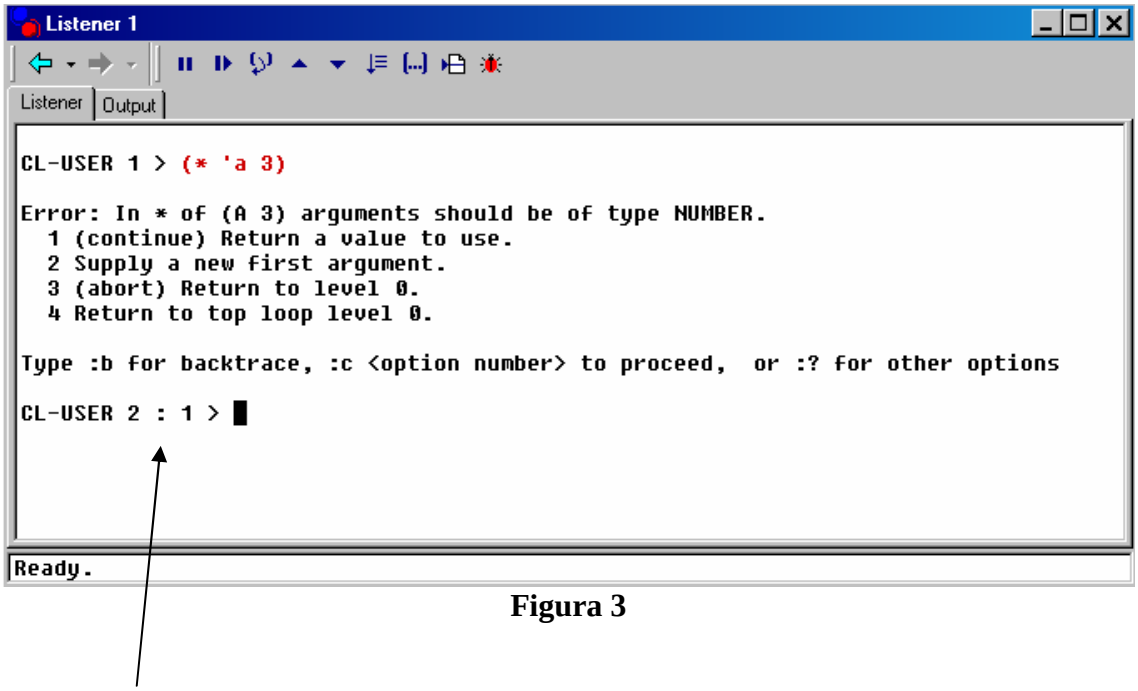


Figura 3

El número 1 mostrado delante del prompt indica que estamos en el Nivel 1 en la depuración.

Aunque las definiciones son recordadas por el intérprete hasta que no se finalice la sesión, no quedan registradas en ningún sitio, salvo el propio intérprete. Existen dos maneras de guardar lo que hemos hecho:
1-Haciendo uso del editor propio de LispWorks.
2-Haciendo uso de ficheros.

Ambas maneras de guardar tienen una diferencia: Si usas el editor no podrás cargar (load) el archivo para que el intérprete lo recuerde.

Por ejemplo, dada una función de suma descrita en la Figura 4, se procederá a guardar el archivo con el editor de LispWork:

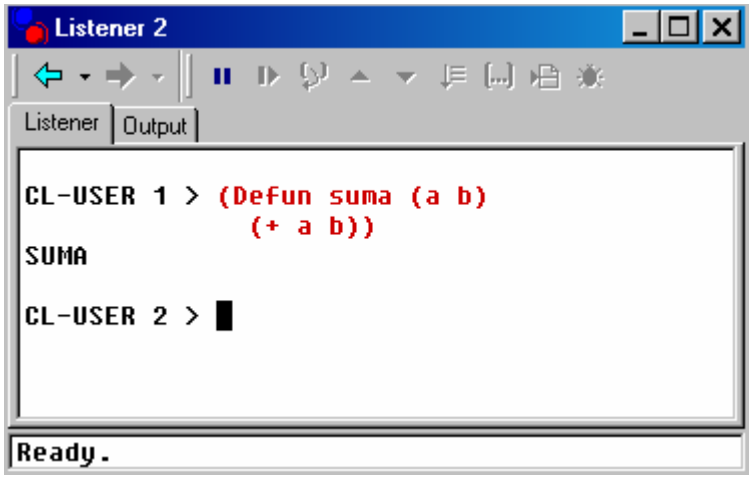


Figura 4

Ahora, nos vamos al menú Tools de la barra de menú y seleccionamos Editor, o bien hacemos clic sobre  en la barra de herramientas como lo muestra la Figura 5:

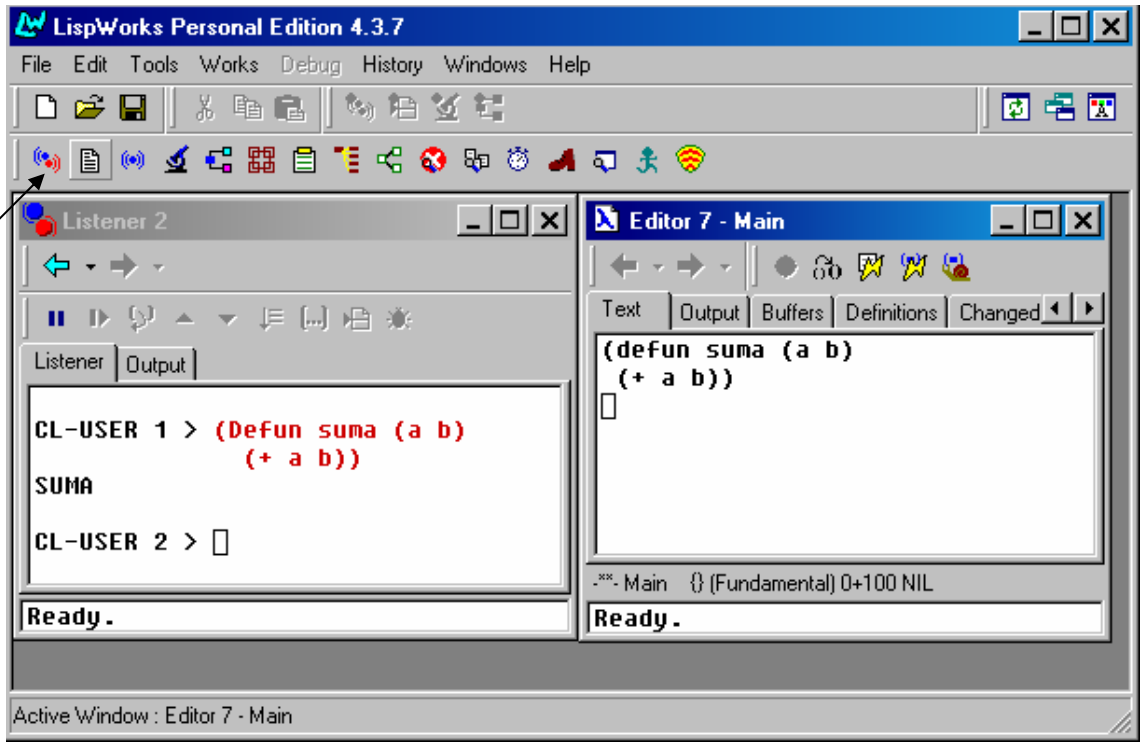


Figura 5

Al final, solo nos vamos al menú File de la barra de menú y le damos Save o Save As, o bien damos clic a  . Luego elegimos el directorio en donde queremos tener el programa y si queremos agregarle un nombre. Le damos Guardar. El archivo que se guarda tiene extensión *.lsp*.

La figura 6 y 7 muestran lo descrito:

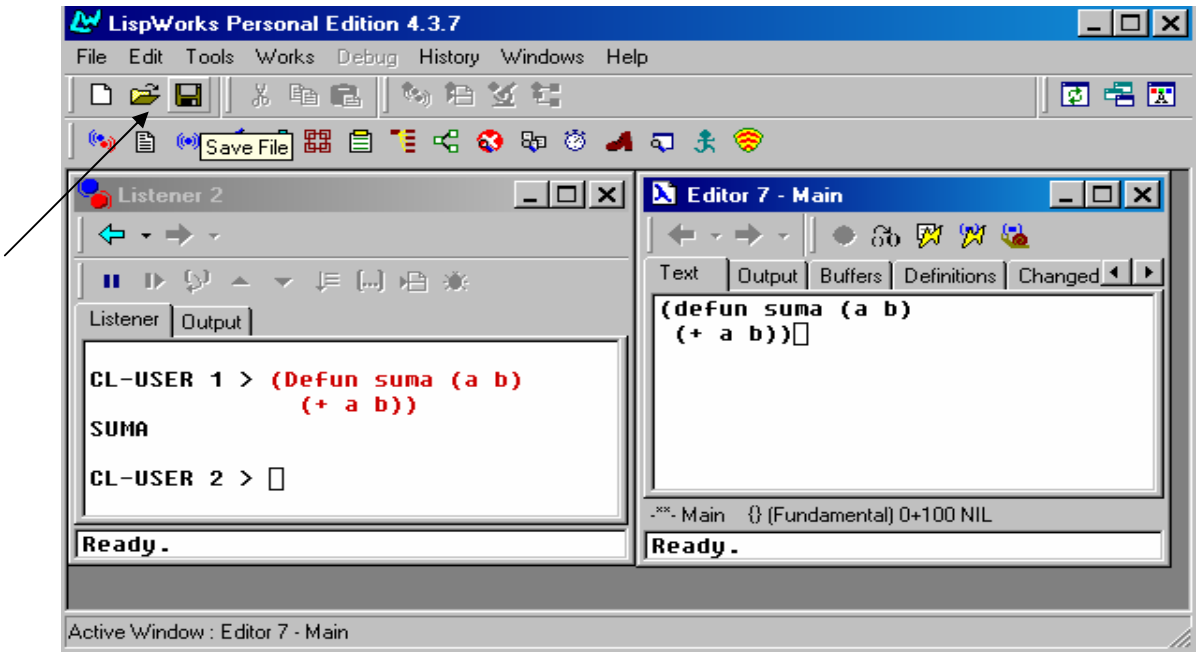


Figura 6

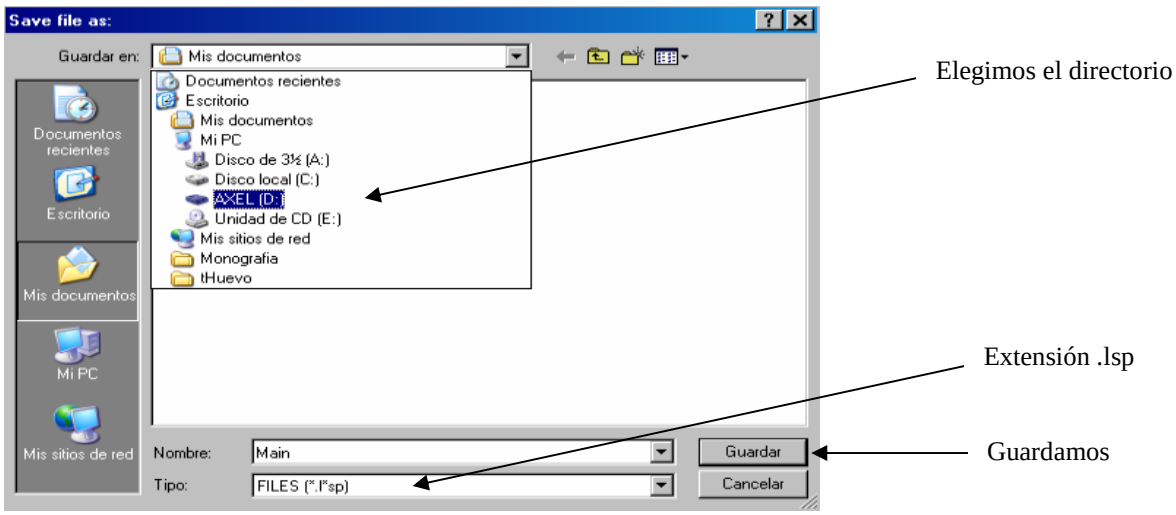


Figura 7

Si queremos tener una copia de algo que hemos hecho aprender al intérprete, necesitas escribirlo en un fichero (se practican más adelante).

Muchos intérpretes modernos, y por ende comerciales, incorporan menús para el manejo y edición de ficheros entre otras nuevas utilidades, pero este intérprete Xanalys LispWorks Personal Edition Version 4.3 "no dispone de facilidades gráficas todavía".

Cargar un fichero en Lisp, para que el intérprete evalué y recuerde todas las expresiones simbólicas que posea, se hace mediante la función LOAD, la cual se abarcó en la unidad 6 (L/E y Ficheros) y se usará para la practica de la misma unidad.
De este modo, podemos escribir el programa en cualquier editor de texto (WordPad por ejemplo) y evaluarlo en el intérprete.

Para terminar con la sesión, se puede hacer mediante la función quit como se muestra:

CL –USER 1 > (quit)

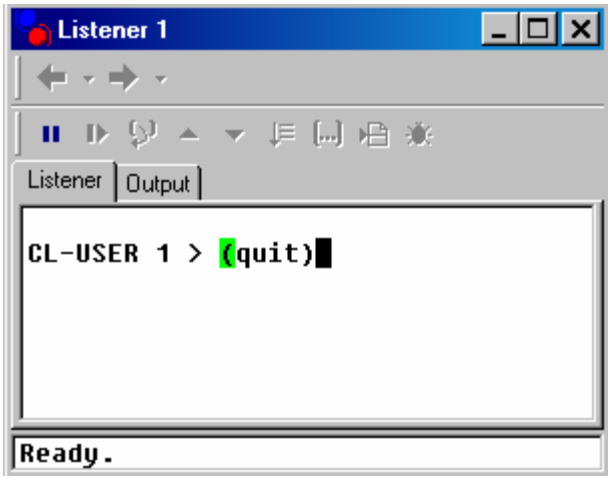


Figura 8

Nota: También, si quieres salir de LispWorks a la hora que quieras solo elige File->Exit.

PRACTICA 2

INTRODUCCION A LISP CON XANALYS LISPWORKS 4.3

Esta segunda sección de prácticas es una introducción al Lisp, empezando por lo más básico (cálculos numéricos). A su vez, se introducen ejercicios sencillos de simulación para que el estudiante entienda como trabaja el intérprete.

Además, el estudiante podrá escribir expresiones numéricas cualesquiera en notación Lisp.

En esta sección se pondrán en práctica los comandos fundamentales de Lisp tales como el control de listas sencillas y su correspondiente resultado.
Por ultimo, el estudiante entenderá de manera practica como trabajan los objetos básicos con los que Lisp trabaja (átomos y listas).

En general, esta práctica abarcará:

♣ Tratamiento de operaciones numéricas que a su vez facilitarán el entendimiento de la forma prefija.

♣ La utilización y significado de los valores lógicos fundamentales (T, NIL, QUOTE).

♣ El uso de algunos comandos fundamentales para el tratamiento y reconocimiento de listas.

♣ Evaluación de expresiones Lisp.

Bibliografía.

*Lisp. Tercera Edición.
Patrick Henry Winston.
Berthold Klaus Paul Horn

*Lisp. El lenguaje de la Inteligencia Artificial.
A. A. Berk

*Introducción al Lisp (El lenguaje básico para la Inteligencia Artificial).
Farreny, H. Masson.

1- Evalúe las siguientes expresiones:

```
a) ( * ( /  ( + 3 3 ) ( - 4 4 ) ) ( * 0 9 ) ).
b) (+ (* 3 (- 5 3)) (/ 8 4)).
c) (* 3 2 2 (- 8 5)).
d) ( ( * )  ( * 2 ( - 4 2 ) )  ( / 8 2 ) ).
```

2- Evalúe las siguientes formas:

```
a) ( SQRT ( ABS( - 71 ( EXPT ( + 2  4 )  4 ) ) ) ).
b) '(  ' +  '1  '( * 2  4 ) ).
c) ( EXPT ( MOD 5  3 ) ( ABS ( - 8  9 ) ) ).
d) ( QUOTE ( NIL  'NIL  T  'T ) ).
e) (LOG (EXPT 2 (- 15 5 4)))
f) (QUOTE (QUOTE (Hello ByeBye))).
```

3- Sea (SETQ A 4 B 6 C 5 X (+ A B) Y (- B C) Z (MAX A C)).
Evalué las expresiones a continuación:

```
a) ( + ( *  C  Z )  B  C ).
b) ( ABS ( + ( * ( -  Z  X  A )  -100 )  B ) ).
c) ( * ( + ( * C  X ) ( - A  Z  C ) )  2  Y ).
```

Ahora se tiene que:
(SETQ M (+ Z A) N (- Y C) P (* 2 Z)).

```
d) ( +  M  Z  X  Y  P  B  N ).
e) ( - ( - ( *  3  Z ) (  /  100  C ) )  A  C  N  P ).
```

4-Evalúe las siguientes expresiones:

```
a) (CONS(CAR '( AXL WIL RICH))(CDR '( ESTE ANTO ALLA)))
b) (CONS(CDAR '((CONS Go Up)))(THIRD'(We Find(All Fine))))
c) (CAR (CDR (CAR (CDR '((a b) (c d) (e f))))))
d) (CAR (CAR (CDR (CDR '((a b) (c d) (e f))))))
e) (CAR (CAR (CDR '(CDR ((a b) (c d) (e f))))))
f) '(CAR (CAR (CDR (CDR ((a b) (c d) (e f))))))
g) (CONS (CAR NIL)(CDR NIL)).
```

5- Evalúe las siguientes formas:

```
a) (CDR(CAR(CDR(CAR '(( D (E F)) G (H I))))))
b) (SETQ A '(+ 3 6)))
    b.1) (CDR A)
    b.2) (CAR(CDR A))
    b.3) (CAR(CDR(CDR A)))
```

6- Escriba secuencias de CAR y CDR para extraer el símbolo MAPACHE de cada una de las siguientes expresiones:

```
a) (oso gato mapache ardilla)
b) ((oso gato) (mapache ardilla))
c) (((oso) (gato) (mapache) (ardilla)))
d) (oso (gato) ((mapache)) (((ardilla))))
```

PRACTICA 3

LISTAS CON XANALYS LISPWORKS 4.3

Esta tercera sección de prácticas es una introducción más a fondo en el tratamiento o manejo de listas en Lisp.

Se evaluaran ejercicios con los distintos tipos de lista estudiados en la unidad 3 y se observaran las diferencias entre un tipo u otro.

Además, el estudiante pondrá en práctica la evaluación de las listas con los distintos comandos, la creación de pequeñas listas de datos y sus correspondientes consultas o asignaciones una vez creadas.

En general, en esta práctica se estudiará:

♣ La evaluación, creación y tratamiento de las listas con los diferentes comandos estudiados en la unidad 3.

♣ Predecir el resultado de las formas que llaman a funciones CAR, CDR, etc., y sus correspondientes abreviados.

♣ Dados ciertos datos, escribir la forma con llamadas a función GET, PUT para producir las salidas esperadas.

♣ Llamar a primitivas que te permitan, dada una lista, determinar su longitud, el último elemento, el elemento de una determinada posición, si un elemento pertenece a la misma, etc.

Bibliografía.

*Lisp. Tercera Edición.
Patrick Henry Winston.
Berthold Klaus Paul Horn

*Lisp. El lenguaje de la Inteligencia Artificial.
A. A. Berk

*Internet:
www.itba.edu.ar

1- Utilizando las funciones para acceder a los elementos de una lista obtener expresiones para extraer el símbolo León y fútbol de las siguientes listas:

```
a) (Managua Chinandega Rivas León Boaco)
b) ((Managua) (Chinandega Rivas León) Boaco)
c) (Managua (Chinandega (Rivas León Boaco)))
d) (deportes (Béisbol tenis) ((fútbol) billar))
```

2- Evaluar las siguientes expresiones:

```
a) (CONS(SECOND '(leo mana china))(CONS(LIST(CAR '(1 2 3 4))
   (CDR '(A B C D)))'2)).

b) (LIST (LIST '(ca ce) (LAST '(0 a 1 b ci) (THIRD '(5 4 3 2 1)))
   (nth 3 '(pa pe pi po pu))).

c) (LIST (APPEND '(H O L A) '(M U N D O)) (CDDR '(sal pan arroz
    pollo)) (CAR '(buen mal)) (CADDR '(desayuno recreo almuerzo
    cena))).

d) (LIST '(¿como estas?)(NTH 0 ' (bien mal rematado))(APPEND
   (CAR '((estas) estoy))(CDR '(entiendo entendiendo lisp)))).

e) (LENGTH (CONS (CAR '(verdad mentira falso)) (LIST* 'es 'muy
    '(facil)))).

f) (CDR(LIST(SUBSEQ '(z y x w v)0 2)(CONS '(a e i) '(o u))).
```

3- Sea (SETQ paises '((Nicaragua . Managua) (Italia . Roma)(España . Madrid))).
Evaluar la expresión siguiente:

```
(LIST 'Nicaragua 'Italia ‘España (SUBLIS paises '(Nicaragua
Italia España)) (NTH 1 '(eran son serán)) (CAR (NTHCDR 1(CDR
'(pueblos estados capitales barrios)))) (CONS 'de (CONS ‘estos
(CONS 'paises NIL))))
```

4- Sean las siguientes listas:

```
Palabras = (grande bonito feliz humedo)

Sinonimos = (alto bello contento mojado)

Antonimos = (pequeño feo triste seco)

Ingles = (big beautiful happy humid)
```

a) Definir las siguientes listas asociativas:

1. Palabras-Sinonimos.
2. Palabras-Antonimos.
3. Palabras-Ingles.

b) Una vez realizado el ejercicio anterior, cual es resultado de las siguientes expresiones:

```
i.(CDR(ASSOC grande Palabras-Ingles)).
ii.(CDR(ASSOC feliz Palabras-Sinonimos)).
iii.(FIRST(ASSOC humedo(ACONS pal anto Palabras-Antonimos)))
iv.(LAST(CONS(LENGTH(ACONS lenguaje Ingles Palabras Ingles))
   (CONS '(agregando)(cons '(palabras)(list '(ala) '(Lista-
    Asoc))))
```

5- Escribir tres P–listas Unan-León, José, ISI que recojan los siguientes datos:

Unan-León	José	ISI
Edad 160	Edad 22	AÑOS:4
Ubicación León	Universidad: Unan-León	numprofesores:23
Rector: RigobertoSampsom	Carrera ISI	ofrece: Unan-León
		Laestudio:Jose

5.1- Consultar las listas creadas en el ejercicio anterior para obtener:

1. La edad de José.

2. El nombre del Rector de la Unan-León.

3. Ubicación de quien ofrece la carrera de ISI.

4. La edad de la universidad donde José estudio ISI.

5. El número de profesores de la carrera de José.

6. Asignar a la edad de José 23.

7. Cambiar el número de profesores de la carrera de José para que ahora sean 28.

PRACTICA 4

DEFINICION DE FUNCIONES CON XANALYS LISPWORKS 4.3

Esta cuarta sección de prácticas es una de las más importantes ya que se practicara la declaración de funciones las cuales son unos de los elementos más importantes en Lisp.

El estudiante se inmiscuirá en la definición de significados funcionales lo cual es sumamente importante ya que a partir de aquí se dará cuenta del porque Lisp es llamado programación funcional.

Además, el estudiante aprenderá a declarar funciones sencillas con los comandos vistos en las prácticas y unidades anteriores.

En general, en esta práctica se estudiará:

♣ La evaluación y creación de funciones sencillas en Lisp, así como la importancia de su utilización en la práctica.

♣ Un uso mas profundo de las primitivas estudiadas anteriormente y la aplicación de las mismas en las funciones.

Bibliografía.

*Lisp. Tercera Edición.
Patrick Henry Winston.
Berthold Klaus Paul Horn

*Lisp. El lenguaje de la Inteligencia Artificial.
A. A. Berk

*Internet:
www.itba.edu.ar

1- Usando los primitivas estudiadas en las practicas y unidades anteriores, resuelva los siguientes ejercicios:

a) Escribir una función POLI que calcule el valor del polinomio de segundo grado $ax^2 + bx + c$.
Por ejemplo: (Poli 4 3 2 2) → 24

b) Defina una función llamada INTERCAMBIO que devuelva los elementos de una lista de dos elementos pero en orden inverso.
Por ejemplo:
(SETF razas '(blancos negros))
(Intercambio razas) → (negros blancos)

c) Defina un procedimiento llamado ROTAR-IZQUIERDA, que tome una lista como argumento y devuelva una lista nueva en la cual el primer elemento aparezca en el último lugar.
Por ejemplo:
(Rotar-Izquierda '(all rich willi) → (rich willi all)
(Rotar-Izquierda(Rotar-Izquierda '(all rich willi) →
→ (willi all rich)

d) Defina ROTAR-DERECHA de manera similar al ejercicio anterior, solo que la rotación debe ser en la otra dirección.

e) Haciendo conversiones entre grados Fahrenheit y grados Celsio, se sabe que -40º Fahrenheit es igual a -40º Celsio. Se tienen las siguientes fórmulas de conversión:

C=(F+40)*5/9-40
F=(C+40)*9/5-40

Usando esas fórmulas, defina dos funciones de conversión FAHR-A-CEL y CEL-A-FAHR.
Por ejemplo:
(Fahr-a-Cel 32) → 0
(Fahr-a-Cel 98.6) → 37.0
(Fahr-a-Cel 212) → 100
f) Asigne a L la siguiente lista: barco, bicicleta, cuadraciclo, avión y KM-3. Ahora, defina una función que pase el tercer elemento (cuadraciclo) a ser el primero de la lista.

g) Defina una función llamada SALIDA que calcule el determinante de una matriz 2*2. Para ello, supóngase una matriz ((a1 a2) (b1 b2)) donde (a1 a2) y (b1 b2) son listas numéricas. El determinante se obtendría multiplicando a1 por b2, a2 por b1 y el resultado de ambos se resta.

PRACTICA 5

PREDICADOS Y ESTRUCTURAS DE CONTROL CON X. L. 4.3

Esta quinta sección de prácticas contiene ejercicios de evaluación y programación de funciones para el manejo de números y listas usando los diferentes tipos de predicados y estructuras de control estudiadas en la unidad 4.

Con esta práctica se termina de estudiar las funciones básicas para pasar a estudiar las estructuras de control más importantes, como son las condicionales, la definición de variables locales, etc.

El alumno aprenderá a escribir expresiones lógicas o condicionales en notación LISP. También sabrá predecir las salidas correctas por cada expresión tomando en cuenta la sintaxis de cada una.

En general, esta práctica abarcará:

♣ El continuo estudio de tratamiento de listas y funciones.

♣ La escritura de cualquier expresión lógica o condicional en Lisp.

♣ La programación de funciones con argumentos tomando en cuenta las distintas condiciones que se le presenten.

♣ La correcta predicción de la salida para funciones que realizan comparaciones entre números o símbolos (=, <=, etc.).

♣ La evaluación de expresiones teniendo en cuenta las salidas no-nil de ciertos predicados.

Bibliografía.

*Lisp. El lenguaje de la Inteligencia Artificial.
A. A. Berk

* Lisp. Tercera Edición.
Patrick Henry Winston.
Berthold Klaus Paul Horn

*Internet:
www.publispain.com
www.itba.edu.ar

1. Evaluar las siguientes expresiones:

a) (OR(<(THIRD '(a 2 3 5 b))3)(NUMBERP(CAR '(a 1 2 3)))
'(verdadero)(EQL 5 5)(LISTP())).

b) (AND(> 9 8 2 0)(PLUSP(SECOND '(1 2 3)))(EQ 'wa 'wa)
(= 2 2.0 2.00) 'todobien).

c) (OR(AND(OR(ZEROP 2)(ENDP(CDR(1 2 a b))))(<= 1 0 2))(CAR((1 2)
a b))).

d) (NOT(OR(AND(NULL(ware))(SYMBOLP(axeru)))(=/ 4 4.0 4.1))).

e) (AND(EQUAL 0.0 0.00)(>= 3 2.50 2.00 1 0)(AND(LISTP
'(1 0 -1 -2))(ODDP(+ 3 2)))).

f) (OR(NOT(NOT(OR())))(MINUSP(CAR(1 2 3)))(EQ '(wa) 'wa)
(OR T ware)(EVENP (/ 6 3))).

g) (EQUAL(AND T(ODDP(* 2 8))NIL)(NOT(OR NIL T wa 12))).

h) (WHEN(NUMBERP(CAR(CDR '(A 1 B 2 C 3)))) 'quefue
(CONS 'es-numero(CONS ‘segundo-elemento NIL))).

i) (UNLESS(=(REM (CAR ‘(26 27 30 41 50))3)0)(SETQ var
'(primer elemento))(SETQ A '(no es div))(SETQ B '(por 3))
(LIST var(CONS a b))).

j) (IF(LISTP(CAR(CDR(CDR '(1 2 (ware) 3 4)))))(APPEND
'(tercer)(CONS 'elemento '(es lista))) '(tercerelenm no es
lista)).

2. Realizar las siguientes funciones:

A. Definir una función que tome un año del calendario y devuelva si es bisiesto o no. Un año es bisiesto si es divisible por 4, a menos que sea divisible por 100 y no por 400.

B. Definir una variable que contenga los siguientes METALES con sus respectivas densidades: HIERRO 7.8, COBALTO 8.9 Y NIQUEL 8.9. Además, definir otra variable llamada GASES-NOBLES que contenga los siguientes gases: NEON, ARGON Y HELIO. Definir una función DESCRIPCION que al leer una expresión X retorne:

- **Una lista de la forma (Metal de Densidad N), si X es un metal de la lista METALES y N es la densidad de X.**
- **GAS NOBLE, si X esta en la lista de GASES-NOBLES.**
- **NO SE TIENE INFORMACION, en otros casos.**

C. Escribir una función llamada operación que reciba dos argumentos numéricos. Si el primero es mayor que el segundo que calcule el cubo del primero menos el duplo del segundo, si es lo contrario que calcule el cuádruplo del segundo aumentado en uno más tres veces el primero disminuido en ocho. Si algún numero es cero mostrar mensaje No-hay-operación y si son iguales hacer la resta y mostrar mensaje Resta de Números-iguales siempre-da 0.

D. Realizar una función llamada Total-pagar que reciba dos argumentos. El primer argumento será el modelo de un carro y el segundo será la cantidad de carros a comprar. El precio del Yaris es $8000, Corolla $13000, Hillux $35000. Si cantidad <= 2 no habrá rebaja. Si cantidad es >2 y <5, el modelo Yaris rebaja el 20%, Corolla rebaja el 25% y Hillux 30%. Si cantidad es >= 5, el modelo Yaris rebaja el 25%, Corolla rebaja el 30% y Hillux 35%. Si no se corresponde con ningún modelo anterior mostrar mensaje Modelo No-Valido.

E. Defina una función que una los dos extremos de una lista utilizando la primitiva Case. Por ejemplo la salida debería ser (Que Helado) si la lista fuera (Que sabroso esta el helado).

PRACTICA 6

ITERACION Y RECURSIÓN CON XANALYS LISPWORKS 4.3

Una de las características de Lisp es su gran facilidad de programar recursivamente. En esta sexta sección de prácticas se introduce la recursividad desde lo más sencillo, para aprender de forma sólida los conceptos básicos.

En este apartado de prácticas se manejarán las construcciones y conceptos referentes a la recursión, creación de variables y funciones locales e iteración general en Lisp. Se tratará de profundizar en la programación recursiva para ser capaces de solucionar problemas cada vez más complejos.

Además, el alumno razonará y empleará técnicas de corrección de programas que se implementen o propongan, dará una explicación de lo que debe hacer una función, escribirá una forma con sintaxis correcta de Lisp y que cuando se invoque produzca el resultado deseado.

En general, en esta práctica se estudiará:

♣ Las sintaxis básicas para iterar con sus correspondientes características.

♣ Como programar funciones de argumento numérico y de listas definidas recursivamente.

♣ Razonar acerca de la terminación y corrección de los procesos recursivos.

Bibliografía.

*Lisp. Tercera Edición.
Patrick Henry Winston.
Berthold Klaus Paul Horn

*Lisp. El lenguaje de la Inteligencia Artificial.
A. A. Berk

*Lisp. The language of Artificial Intelligence.
Collins London.

Internet:
www.Google.com

1- Evaluar las siguientes expresiones:

a)
```
(DO ((X '(A B C) (CDR X))
     (Y '(D E F G H) (CDR Y))
     (Z '(A B)))
    ((NULL X) (APPEND Y Z)))
```

b)
```
(DO ((X '(A B C) (CDR X))
     (Y '(D E F G H))
     (Z))
    ((NULL X) (CONS Y Z))
     (CONS X Y))
```

c)
```
(DO* ((N 1 (1+ N))
      (M N (* 2 M)))
      ((> M 10) M N))
```

d) **Dado el siguiente programa explique su salida cuando se le pasan los parámetros correspondientes.**

```
(DEFUN times (a b)
           (COND((zerop a) 0)
                (T (+ b (times (- a  1) b)))))
```

e) **Dado el siguiente programa recursivo, explique su ejecución cuando se le pasan los argumentos correspondientes.**

```
(DEFUN primer-ultimo (lista)
     (CONS (FIRST lista)
           (ultimo (CDR lista))))

(DEFUN ultimo (lista)
  (IF (CDR lista) (ultimo (CDR lista)) lista))
```

2- Definir una función iterativa que calcule el factorial de un número. Haga uso de DO o de DOTIMES.

3- La función LENGTH devuelve la longitud de una lista. Definir dos funciones que hagan lo mismo. Una de las funciones será recursiva y la otra función será iterativa.

4- La función REVERSE retorna una lista inicial pero en orden inverso. Definir una función recursiva que haga lo mismo.
Ejemplo: (inverso '(ab cd ef gh)) → (gh ef cd ab).

5- Escriba una función de nombre total-impares que reciba una lista cualquiera y de cómo resultado los números impares que posea.

6- Definir una función que una dos lista de cualquier tamaño y retorne una lista con los elementos de ambas.
Ejemplo: (unir '(ab cd ef) '(wx yz)) → (ab cd ef wx yz).

7- Se le llama recursión a aquellos programas que se llaman entre sí para realizar cierto trabajo. Se sabe que las funciones EVENP y ODDP retornan, T si el número que se pasa como argumento es Par y NIL si el número es Impar, respectivamente. Definir dos funciones que hagan lo mismo y se llamen entre sí para evaluar un número determinado.
Ejemplo: Se tiene las funciones (num-par n) , (num-impar n).
>>> (num-par 10) → T
>>> (num-impar 6) → NIL

8- Construir una base de datos bibliográfica. Para ello, definir la función AGREGAR-LIBRO que contenga referencia, autor, título y editorial. Debe añadir el libro especificado a la base datos y tiene que retornar referencia. Se utilizará la lista de propiedades LIBRO-N que contiene autor, título y editorial del libro de referencia N y la variable global LIBROS que contiene la lista de propiedades almacenadas.
Por ejemplo, la función AGREGAR-LIBRO tendrá que hacer lo siguiente:

```
(SETF (SYMBOL-PLIST 'Libro-1)
'(AUTOR "Juan Venado"
TITULO "Sueños Líquidos"
EDITORIAL "Pancha"))
(SETQ Libros '(Libro-1))
```

Si se hiciera la llamada (Get 'libro-1 'editorial) este retornaría "Pancha".

9- Tomando en cuenta el ejercicio anterior, definir una función BUSCAR-POR que contenga TIPO el cual puede ser editorial, autor o título, VALOR, que contiene el valor del TIPO. La función BUSCAR-POR buscará en la variable global LIBRO y devolverá el número del libro al que pertenece el TIPO.

10- Escriba una función llamada *POSICIÓN-CAMBIO* que reciba tres argumentos los cuales serán una lista, un elemento (elemento viejo) de la lista que será reemplazado por el otro elemento (Elemento nuevo). Dicha función dirá en que posición de la lista fue el cambio de lo contrario mostrará No-hubo-cambio. Para ello, dicha función tendrá que llamar a una función POSICION-ATOMO, la cual indica donde se encontró el átomo, y a una función CAMBIA la cual cambiará el átomo y retornará la lista con el nuevo átomo. La función POSICION-ATOMO deberá llamar a una función BUSCAR-ATOMO que es la que encontrará el átomo en la posición encontrada por POSICION-ATOMO.

Por ejemplo:

(Posición-Cambio ‘(a b c d e f) ‘c ‘z) ---> (c cambiado por z en posición 3)

PRACTICA 7

LECTURA/ESCRITURA Y FICHEROS CON XANALYS LISPWORKS 4.3

En esta séptima sección de prácticas se exponen y aplican las construcciones Lisp que permiten la comunicación con el mundo exterior (E/S), así como algunas que permiten el control y uso de ficheros.

Esta práctica es de mucha importancia porque es con las E/S que el programador Lisp le enseña al usuario como interactuar con un sistema cualquiera y a la vez como se guardan los datos en archivos para su posterior uso.

Además, se aprenderá algo nuevo como es el manejo de datos de tipo carácter o cadena lo cual no se había abarcado en las prácticas anteriores.
También se estudiará el uso y manipulación de ficheros los cuales son muy conocidos en la casi todos los lenguajes de programación existentes.

En general, en esta práctica se aprenderá a:

♣ Manejar datos tipo carácter y cadena.

♣ Predecir la salida de expresiones que llaman a funciones que realizan la E/S y operaciones sobre ficheros.

♣ Escribir expresiones para escribir y leer utilizando una salida y entrada particular.

♣ Escribir funciones que se comuniquen con el usuario a través del teclado y la pantalla.

♣ Escribir funciones que se comuniquen con ficheros del sistema.

Bibliografía.

*Lisp. Tercera Edición.
Patrick Henry Winston.
Berthold Klaus Paul Horn

*Lisp. El lenguaje de la Inteligencia Artificial.
A. A. Berk

*Internet:
http://.go.to/inteligenciartificial

1- Evaluar las siguientes expresiones e indicar los efectos que estas producen durante su evaluación:

```
a)(PRINT (PRIN1 (PRINT (PRIN1 (* 3 5))))).

b)(PRINT (LIST 'Tu 'nombre 'es (CADR(PRINT '(¿Como te
llamas?))(READ)))).

c)(FORMAT T "¡Que onda! ~S" (+ 4 (PRINT (PRINT (PRINT (PRINT
2)))))).

d)(FORMAT NIL "¡Como estas! ~S" (+ 8 (PRINT (PRINT (PRINT (PRINT
9)))))).
```

2- Definir una función sin argumentos que pregunta al usuario que introduzca un número cualquiera. Si el usuario introduce el número retornara el mismo número pero si introduce algo distinto de un número que repita la pregunta hasta que sea un número.

3- Escribir un programa el cual pida el nombre del usuario, su apellido, su edad y lugar de nacimiento y sean capturados. Preguntar si los datos son correctos y si los son retornar los datos como una lista. En caso de no serlo volver a pedir los datos.

4- Dada la función del factorial que se muestra a continuación, elaborar una segunda función que llame a factorial la cual muestre un mensaje de error y NIL si:
a) el dato introducido no es un número.
b) el número no es entero.
c) el número es negativo.

```
(DEFUN factorial (n)
     (IF (= n 0) 1 (* n (factorial (- n 1)))))
```

5- Sea el fichero articulos.dat donde se almacenan electrodomésticos y su respectivo precio.
¿Qué resultado producen las siguientes expresiones?

a.
```
(WITH-OPEN-FILE(ART "ARTICULOS.DAT" :DIRECTION :INPUT)
          (READ ART)(READ ART))
```

b.
```
(WITH-OPEN-FILE(ALUMNO "CLASES.DAT" :DIRECTION :OUTPUT)
          (FORMAT ALUMNO " NOMBRE ~% NOTA"))
```

c.
```
(WITH-OPEN-FILE(ALUMNO "CLASES.DAT" :DIRECTION :INPUT)
            (LOOP (READ ALUMNO)))
```

d.
```
(WITH-OPEN-FILE(ALUMNO "CLASES.DAT" :DIRECTION :INPUT)
                 (LOOP(READ ALUMNO NIL)))
```

6- Definir una función llamada Deptoleon que lea del fichero Reos.dat y escriba en el fichero Deptoleon.dat los reos que pertenecen al departamento de León. Nota: Primero crear el fichero Reos.Dat.

Supongamos el fichero Reos.dat con el siguiente contenido:

((Nombre) (Departamento) Comportamiento Edad Condena Tiempo-Prisión)

```
((Marcos Toruño) Managua Excelente 42 5 3)
((Alvaro Zapata) Leon Deficiente 51 8 2)
((David Rivas) Leon Regular 39 3 2)
((Abelardo Alvarado) Chinandega Muy-Bueno 48 15 8)
((Norman Ruiz) Leon Muy-bueno 24 10 4)
((Mario Bonilla) Managua Buena 30 20 5)
```

7- Defina una función llamada LIBRE que lea del fichero Reos.dat y escriba en el fichero SALE.dat los reos a los cuales es posible darles la libertad de lo contrario escriba en el fichero QUEDA.dat si no cumple las condiciones. Si el comportamiento es Excelente, Muy-bueno, o Bueno y el reo a cumplido la mitad de sus condenas se le otorgará su libertad.

CONCLUSIÓN

En la elaboración de este soporte para la asignatura de Inteligencia Artificial y Sistemas Expertos no solo se hizo uso de toda la información conseguida en la red sino que se utilizó la experiencia que obtuvimos como estudiantes de 5to. Año de Ingeniería en Sistemas de Información para encontrar un orden lógico con las unidades que se presentan y también para las prácticas.

La materia de Inteligencia Artificial y Sistemas Expertos es bastante extensa en lo teórico y practico ya que son muchos los aspectos que se estudian. Con este trabajo monográfico se eligieron los temas y el orden de las unidades muy cuidadosamente en base a los niveles de dificultades presentados.

En el soporte se colocaron los aspectos más generales e importantes por cada unidad y se abarcó, con las 7 unidades, un orden satisfactorio y globalizado sobre la Inteligencia Artificial, los Sistemas Expertos y el lenguaje Lisp.

La forma en que se presenta la información en este trabajo facilitará la enseñanza al profesor y el aprendizaje de la asignatura por parte de los estudiantes.

ANEXOS

Resumen de Sintaxis

Se presenta a continuación un resumen de determinados elementos que forman parte de la sintaxis utilizada en Lisp:

- **(** Sirve para comenzar una lista de elementos. +
- **)** Cierra una lista de elementos.
- **’** Un acento agudo (llamado también apostrofe) seguido de una expresión, es una abreviatura de (quote expresión).
- **;** Es el carácter para escribir comentarios.
- **“** Sirven para escribir cadenas de caracteres.
- \ Sirve de carácter de escape. Hace que el siguiente carácter se trate como si fuese una letra en lugar de tomar su significado sintáctico.
- **|** Se usan en pareja, para rodear el nombre de un símbolo que contiene muchos caracteres especiales.
- **#** Se coloca al comienzo de una estructura sintáctica.
- **‘** Sirve para hacer plantillas, muy útiles, especialmente en la construcción de macros.
- **,** Se usan dentro de las expresiones construidas con ‘.
- **:** Sirve para indicar a que package pertenece un símbolo.

Otros Comandos Lisp

Funciones Numéricas Predefinidas

(+ n1 n2 ... nN): Devuelve el valor de la suma n1 + n2 + ::: + nN. Si N = 0, da 0.

(- n1 n2 ... nN): Devuelve el valor de n1 ¡ n2 ¡ ::: ¡ nN. Si N = 1, da ¡n1.

(* n1 n2...nN): Devuelve el valor del producto n1:n2:::nN. Si N = 0, da 1.

(/ n1 n2): Devuelve el valor de dividir n1 por n2.

(/ n): Es lo mismo que (/ 1 n); es decir, devuelve el inverso de n.

(Sqrt n): Tiene significado funcional la función raíz cuadrada.

(Expt base expo): Tiene significado funcional la función exponencial de base cualquiera.

Log: Tiene significado funcional la función logaritmo. Si tiene un solo argumento es logaritmo natural y si tiene un segundo será su base.
Sintaxis: (LOG n), (LOG n1 n2).

(Abs n): Tiene significado funcional la función valor absoluto.

(Float(Fun-Num n1 n2 ... nN)): Convierte el resultado en un número entero.

(Round(Fun-Num n1 n2 ... nN)): Convierte el resultado en un número de punto flotante.

(Mod n1 n2): Tiene significado funcional predefinido la función modulo.

(Max n1……..nN): Tiene significado funcional la función máxima de cualquier número de argumentos.

(Min n1……nN): Tiene significado funcional la función mínimo de cualquier número de argumentos.

+1: Tiene significado funcional la función sucesor.

=1: Tiene como funcional la función predecesor.

Estructuras de Control

(SET ’simb s): Asigna, destruyendo cualquier asignación previa, el valor de la expresión s al símbolo simb. Devuelve el valor de s.

(MAKUNBOUND simb): borra el valor asignado a simb.

Funciones Trigonométrica y Matemática

(SIN n): Devuelve el seno de n radianes.

(COS n): Devuelve el coseno de n radianes.

(TAN n): Devuelve la tangente de n radianes.

(ASIN n): Devuelve el arco seno de n (expresado en radianes).

(ACOS n): Devuelve el arco coseno de n (expresado en radianes).

(ATAN n): Devuelve el arco tangente de n (expresado en radianes).

Descripción de Símbolos

(DESCRIBE s): Devuelve información sobre el objeto s.

(SYMBOL-NAME simb): Devuelve el nombre del símbolo simb.

(SYMBOL-VALUE simb): Devuelve el valor asociado al símbolo simb.

(SYMBOL-FUNCTION simb): Devuelve la función asociada al símbolo simb.

(SYMBOL-PLIST simb): Devuelve la P–lista asociada al símbolo simb.

Funciones de Evaluación

(PROG1 s1... sN): Evalúa sucesivamente las expresiones s1,..., sN y devuelve el valor de s1.

(PROG2 s1...sN): Evalúa sucesivamente las expresiones s1,..., sN y devuelve el valor de s2.

Funciones de Listas

(Remove s list): Devuelve una nueva lista que contiene todos los elementos de la lista original, menos el objeto indicado.

(MAKE-LIST n): Devuelve una lista formada por n elementos NIL.

(MAKE-LIST n :INITIAL-ELEMENT s): Devuelve una lista formada por n elementos s.

(REVERSE L): Devuelve una lista, con los elementos L en orden inverso. Función REV que actúe como REVERSE pero a todos los niveles.

(NREVERSE L): Devuelve la lista L, con sus elementos en orden inverso, a diferencia que es destructiva.

(BUTLAST L n): Devuelve una lista sin los n-últimos elementos de L.

(SUBST sn sa s): Devuelve la expresión s, en la que se ha sustituıdo sa (expresión antigua) por sn (expresión nueva) a todos los niveles.

(Member S Lista): Verifica que su primer argumento sea un elemento de su segundo y devuelve lo que queda de la lista al encontrar dicho elemento.

(Search Cad1 Cad2): Determina si una cadena esta contenida en otra y devuelve la posición en donde comienza la primera cadena.

Funciones de Modificación Física

(RPLACA L s): Devuelve la lista L, en la que se ha reemplazado el CAR por s y L queda alterada.

(RPLACD L s): Devuelve la lista L, en la que se ha reemplazado el CDR por s y L queda alterada.

(NCONC L1 ... lN): Devuelve L1, pero modificada concatenando físicamente todas las listas Li. Además, modifica cada lista concatenando físicamente las siguientes.

(DELETE s L): Devuelve la lista L, en la que se han borrado las estancias de primer nivel de la expresión s y L queda alterada.

(C...R L): Es una combinación de CAR y CDR hasta 4 niveles.

El Rastreador

(TRACE fn1 ... fnN): Permite rastrear las funciones fn1,..., fnN mostrando la acción de las funciones fn1,..., fnN cada vez que actúan.

(UNTRACE fn1 ... fnN): Elimina el efecto de TRACE de las funciones fn1,..., fnN.

(UNTRACE): Elimina el efecto de TRACE de toda función que lo tenga. Devuelve la lista de las funciones con TRACE.

Ejecución paso–a–paso

(STEP s) Evalúa la expresión s paso a paso, de forma que se puede controlar el proceso a voluntad.

Funciones de Aplicación

(MAPC fn l): Aplica la función fn tomando como argumento los primeros elementos de la lista l, después toma como argumento los segundos elementos de la lista y así hasta que la lista se termina.

(MAPLIST fn l1...lN): Devuelve la lista con los resultados de aplicar fn a las li, luego a los CDR de las li, luego a los CDDR de las li, hasta que una se termina.

(MAPCAR ‘Fun L): Toma una función y una o más listas, y aplica dicha función sobre cada elemento de la lista, produciendo una nueva lista resultante.

(EVERY fn l): Aplica la fn a los elementos de l hasta que una aplicación de NIL, o hasta que la lista se termine, en cualquier caso, devuelve la ultima aplicación.

(SOME fn l): Aplican la fn a los elementos de l hasta que una aplicación de distinto de NIL o hasta que la lista se termine, en cualquier caso, devuelve la ultima aplicación.

(RETURN s): Devuelve el valor de s, y detiene el ciclo si dentro de un PROG o un LOOP, cuando no lo esta da error.

Funciones Sobre el Sistema

(Exit): Termina la sesión Lisp.

(Time s): Devuelve el tiempo usado para evaluar la expresión s.

(Dos cadena): Ejecuta el comando cadena del sistema operativo DOS.

(Push item lugar): La forma que se pasa como segundo argumento (lugar) ha de evaluarse dando lugar a una lista, mientras que el primer argumento (item) puede ser cualquier objeto Lisp. El ítem se añade al principio de la lista y esta nueva lista se almacena en lugar de la anterior.

(Pop lugar): Hace exactamente lo contrario que push, retornando el elemento que se extrae de la cabeza de la lista.

APLICACIÓN

Enunciado

Inicialmente se dispone de un tablero de 3 filas y 3 columnas y por tanto con 9 huecos, que se encuentran vacíos. Cada jugador va poniendo alternativamente fichas en uno de los huecos vacíos, un jugador pone X y el otro pone O (en cualquier caso las fichas de un jugador son distintas de las del otro). Gana el primero que consigue colocar 3 de sus fichas en línea, ya sea una fila, una columna o una de las dos diagonales del tablero.

Código Fuente de dicha Aplicación.

Representación de estados

```
(defstruct (estado (:constructor crea-estado)
                   (:conc-name nil)
                   (:print-function escribe-estado))
            tablero  ficha)
```

Escritura de estados

```
(defun escribe-estado (estado &optional (canal t) profundidad)
     (let ((tablero (tablero estado)))
(format t "~% ~15d ~d ~d        0 1 2~% ~15d ~d ~d        3 4 5~% ~15d ~d ~d        6 7
8~%~%"
                (or (nth 0 tablero) ".")
               (or (nth 1 tablero) ".")
               (or (nth 2 tablero) ".")
               (or (nth 3 tablero) ".")
               (or (nth 4 tablero) ".")
               (or (nth 5 tablero) ".")
               (or (nth 6 tablero) ".")
               (or (nth 7 tablero) ".")
               (or (nth 8 tablero) "."))))
```

Estado inicial

```
(defvar *estado-inicial*)
(defun crea-estado-inicial (ficha)
    (setf *estado-inicial* (crea-estado :tablero '(nil nil nil nil nil nil nil nil nil)
    :ficha ficha)))
```

Estados finales

```
(defun es-estado-final (estado)
    (or (es-estado-completo estado)  (tiene-linea-ganadora estado)))

(defvar *lineas-ganadoras* '((0 1 2) (3 4 5) (6 7 8)
(0 3 6) (1 4 7) (2 5 8)
(0 4 8) (2 4 6)))
```

```
(defun es-estado-completo (estado)
    (not (position nil (tablero estado))))

(defun tiene-linea-ganadora (estado)
     (loop for linea in *lineas-ganadoras* thereis
          (es-linea-ganadora linea (tablero estado))))

(defun es-linea-ganadora (linea tablero)
     (let ((valor (nth (first linea) tablero)))
          (when (and (not (null valor)) (equalp valor (nth (second linea) tablero))
               (equalp valor (nth (third linea) tablero))) valor)))
```

Movimientos

```
(defun ficha-opuesta (ficha)
     (if (eq ficha 'x) 'o  'x))

(defvar *movimientos*)
(setf *movimientos* (loop for i from 0 to 8 collect  (list*  'quieres  'ubicar-ficha-en i nil
)))

(defun ubicar-ficha-en (posicion estado)
     (when (null (nth posicion (tablero estado)))
          (let ((nuevo-tablero (loop for i in (tablero estado) collect i)))
            (setf (nth posicion nuevo-tablero) (ficha estado))
            (crea-estado :tablero nuevo-tablero :ficha (ficha-opuesta (ficha estado))))))
```

Función Aplica Movimiento

Aplicable si la casilla no está ocupada.
Estado resultante: colocar la ficha que toca en la casilla especificada

```
(defun aplica-movimiento (movimiento estado)
     (funcall (symbol-function (second movimiento))
          (third movimiento) estado))
```

La función es-estado-ganador

Devolverá verdadero si y solo en el estado que se le pase como argumento, que describe la situación del juego, cuando le toca mover al jugador TURNO, el JUGADOR ha ganado la partida. En caso contrario devuelve NIL.

```
(Defun es-estado-ganador (estado turno jugador)
     (If (tiene-linea-ganadora estado) (not (equalp jugador turno))
         Nil))
```

Generador de juego

```
(Defun juego-inicia-con-ficha (ficha)
        (crea-estado-inicial ficha)
        (list '3-en-raya. 'Empieza 'ficha ficha))
```

Función de evaluación estática

```
(defparameter *minimo-valor* -99999)
(defparameter *maximo-valor* 99999)

(defun f-e-estatica (estado jugador)
     (cond ((es-estado-ganador estado jugador 'Ordenador) *maximo-valor*)
          ((es-estado-ganador estado jugador 'Humano) *minimo-valor*)
          ((es-estado-completo estado) 0)
          (t (- (posibles-lineas-ganadoras estado jugador 'Ordenador)
          (posibles-lineas-ganadoras estado jugador 'Humano)))))

(defun posibles-lineas-ganadoras (estado turno jugador)
     (Loop for linea in *lineas-ganadoras* counting (not (esta (tablero estado)
     (If (eq turno jugador) (ficha-opuesta (ficha estado))
          (ficha estado))
     linea))))

(Defun esta (tablero ficha linea)
     (loop for i in linea thereis (eq (nth i tablero) ficha)))
```

Procedimiento de control de juegos

```
(Defvar *nodo-j-inicial*)
(Defstruct (nodo-j (:constructor crea-nodo-j) (:conc-name nil)
    (:print-function escribe-nodo-j)) estado jugador valor)
(Defun escribe-nodo-j (nodo-j &optional (canal t) profundidad)
     (Format canal "~%Estado : ~a~%Jugador : ~a" (estado nodo-j)
     (jugador nodo-j)))

(Defun crea-nodo-j-inicial (jugador)
     (Setf *nodo-j-inicial* (crea-nodo-j :estado *estado-inicial* :jugador jugador)))

(defvar *procedimiento*)

(Defun juego (&key (empieza-la-maquina? nil)
     (procedimiento '(minimax 10)))
     (Setf *procedimiento* procedimiento)
          (Cond (empieza-la-maquina? (crea-nodo-j-inicial 'Ordenador)
               (If (es-estado-final *estado-inicial*) (analiza-final *nodo-j-inicial*)
                    (jugada-maquina *nodo-j-inicial*)))
                    (T (crea-nodo-j-inicial 'Humano)
                    (If (es-estado-final *estado-inicial*) (analiza-final *nodo-j-inicial*)
                         (jugada-humana *nodo-j-inicial*)))))
```

```
(Defun analiza-final (nodo-j-final)
      (escribe-nodo-j nodo-j-final)
      (Cond ((es-estado-ganador (estado nodo-j-final)
          (jugador nodo-j-final) 'Ordenador) (format t "~&La maquina ha ganado"))
          ((es-estado-ganador (estado nodo-j-final)
          (jugador nodo-j-final) 'Humano)
          (Format t "~&El humano ha ganado"))
          (T (format t "~&Empate"))))

(Defun movimientos-legales (estado)
    (Loop for m in *movimientos* when (aplica-movimiento m estado)  collect m))

(Defun escribe-movimientos (movimientos)
    (Format t "~%Los movimientos permitidos son:")
    (let ((numero 0))
        (loop for m in movimientos do
            (If (= (mod numero 2) 0)
                (Format t "~% ~a (Teclea el numero ~a)" m numero)
                (Format t " ~a ( Teclea el numero ~a)" m numero))
                (Setf numero (+ numero 1)))))

(Defun jugada-humana (nodo-j)
    (escribe-nodo-j nodo-j)
    (Let ((movimientos (movimientos-legales (estado nodo-j))))
        (escribe-movimientos movimientos)
        (Format t "~%Tu turno: ")
        (let ((m (read)))
            (cond ((and (integerp m)
                 (< -1 m (length movimientos)))
        (Let ((nuevo-estado
            (aplica-movimiento (nth m movimientos) (estado nodo-j))))
            (Cond (nuevo-estado
            (Let ((siguiente (crea-nodo-j :estado nuevo-estado :jugador 'Ordenador)))
        (If (es-estado-final nuevo-estado)
            (analiza-final siguiente) (jugada-maquina siguiente))))
            (T (format t "~& El movimiento ~a no se puede usar. " m)
            (jugada-humana nodo-j)))))
            (T (format t "~& ~a es ilegal. " m) (jugada-humana nodo-j))))))

(Defun jugada-maquina (nodo-j)
    (escribe-nodo-j nodo-j)
    (Format t "~%Mi turno.~&")
    (Let ((siguiente (aplica-decision *procedimiento* nodo-j)))
    (If (es-estado-final (estado siguiente))
        (analiza-final siguiente)
        (jugada-humana siguiente))))
```

Sucesores
La función (SUCESORES ESTADO)

Devolverá una la lista de los sucesores del NODO-J.

```
(Defun sucesores (nodo-j)
    (Let ((resultado ()))
    (Loop for movimiento in *movimientos* do
    (Let ((siguiente (aplica-movimiento movimiento
    (estado nodo-j))))
    (when siguiente (push (crea-nodo-j :estado siguiente
    :jugador (contrario (jugador nodo-j))) resultado))))
    (nreverse resultado)))
```

La función (Contrario (Jugador))

Devuelve Humano si jugador es Maquina, en caso contrario devuelve Maquina.

```
(defun contrario (jugador)
    (if (eq jugador 'Ordenador) 'Humano 'Ordenador))
```

Forma de aplicar los procedimientos de decisión
La función (Aplica-Decision Procedimiento Nodo-j)

El resultado de aplicar el PROCEDIMIENTO de decisión al NODO-J. En este caso la descripción del PROCEDIMIENTO de decisión consiste en una lista cuyo primer elemento es el nombre de la función a aplicar (MINIMAX o MINIMAX-A-B) y cuyo segundo argumento es la profundidad de análisis.

```
(defun aplica-decision (procedimiento nodo-j)
  (funcall (symbol-function (first procedimiento))
           nodo-j
           (second procedimiento)))
```

Procedimiento minimax

Devuelve un nodo con el valor estático del estado de NODO-J, si éste es un estado final, se ha alcanzado la profundidad de análisis máxima o no tiene sucesores. En caso contrario el mejor sucesor del NODO-J según el procedimiento minimax.

```
(Defun minimax (nodo-j profundidad)
    (If (OR (es-estado-final (estado nodo-j)) (= profundidad 0))
        (crea-nodo-j :valor (f-e-estatica (estado nodo-j)
        (jugador nodo-j)))
        (let ((sucesores (sucesores nodo-j)))
        (if (null sucesores)
        (crea-nodo-j :valor (f-e-estatica (estado nodo-j)
        (jugador nodo-j)))
        (if (eq (jugador nodo-j) 'Ordenador)
        (maximizador sucesores profundidad)
        (minimizador sucesores profundidad))))))
```

```
(defun maximizador (sucesores profundidad)
    (let ((mejor-sucesor (first sucesores))
    (mejor-valor *minimo-valor*))
    (loop for sucesor in sucesores do
    (setf valor (valor (minimax sucesor (1- profundidad))))
    (when (> valor mejor-valor)
    (setf mejor-valor valor)
    (setf mejor-sucesor sucesor)))
    (setf (valor mejor-sucesor) mejor-valor)
    mejor-sucesor))
(defun minimizador (sucesores profundidad)
    (let ((mejor-sucesor (first sucesores))
    (mejor-valor *maximo-valor*))
    (loop for sucesor in sucesores do
    (setf valor (valor (minimax sucesor (1- profundidad))))
    (when (< valor mejor-valor)
    (setf mejor-valor valor)
    (setf mejor-sucesor sucesor)))
    (setf (valor mejor-sucesor) mejor-valor)
    mejor-sucesor))
```

Para correr el juego lo primero que se hace es llamar a la función: (JUEGO-INICIA-CON-FICHA 'X) si el usuario quiere x o si no pone 0.
Luego se llama la función (JUEGO).

Salida después de ejecutar las funciones:

```
Estado:
. . .    0 1 2
. . .    3 4 5
. . .    6 7 8


Jugador: HUMANO
Los movimientos permitidos son:
(QUIERES UBICAR-FICHA-EN 0) (Teclea el numero 0) (QUIERES UBICAR-FICHA-EN 1) (Teclea el numero 1)

(QUIERES UBICAR-FICHA-EN 2) (Teclea el numero 2) (QUIERES UBICAR-FICHA-EN 3) (Teclea el numero 3)

(QUIERES UBICAR-FICHA-EN 4) (Teclea el numero 4) (QUIERES UBICAR-FICHA-EN 5) (Teclea el numero 5)

(QUIERES UBICAR-FICHA-EN 6) (Teclea el numero 6) (QUIERES UBICAR-FICHA-EN 7) (Teclea el numero 7)

(QUIERES UBICAR-FICHA-EN 8) (Teclea el numero 8)
Tu turno: 4
```

Estado :
. . . 0 1 2
. X . 3 4 5
. . . 6 7 8

Jugador : ORDENADOR
Mi turno.

Estado:
O . . 0 1 2
. X . 3 4 5
. . . 6 7 8

Jugador: HUMANO
Los movimientos permitidos son:
(Quieres UBICAR-FICHA-EN 1) (Teclea el numero 0) (QUIERES UBICAR-FICHA-EN 2) (teclea el numero 1)

(Quieres UBICAR-FICHA-EN 3) (Teclea el numero 2) (QUIERES UBICAR-FICHA-EN 5) (teclea el numero 3)

(Quieres UBICAR-FICHA-EN 6) (Teclea el numero 4) (QUIERES UBICAR-FICHA-EN 7) (teclea el numero 5)

(Quieres UBICAR-FICHA-EN 8) (Teclea el numero 6)
Tu turno: 1

Estado :
O . X 0 1 2
. X . 3 4 5
. . . 6 7 8

Jugador : ORDENADOR
Mi turno.

Estado :
O . X 0 1 2
. X . 3 4 5
O . . 6 7 8

Jugador : HUMANO
Los movimientos permitidos son:
(Quieres UBICAR-FICHA-EN 1) (Teclea el numero 0) (QUIERES UBICAR-FICHA-EN 3) (Teclea el numero 1)

(Quieres UBICAR-FICHA-EN 5) (Teclea el numero 2) (QUIERES UBICAR-FICHA-EN 7) (Teclea el numero 3)

(Quieres UBICAR-FICHA-EN 8) (Teclea el numero 4)
Tu turno: 1

Estado :
O . X 0 1 2
X X . 3 4 5
O . . 6 7 8

Jugador : ORDENADOR
Mi turno.

Estado :
O . X 0 1 2
X X O 3 4 5
O . . 6 7 8

Jugador : HUMANO
Los movimientos permitidos son:
(Quieres UBICAR-FICHA-EN 1) (Teclea el numero 0) (QUIERES UBICAR-FICHA-EN 7) (Teclea el numero 1)

(Quieres UBICAR-FICHA-EN 8) (Teclea el numero 2)
Tu turno: 1

Estado :
O . X 0 1 2
X X O 3 4 5
O X . 6 7 8

Jugador : ORDENADOR
Mi turno.
Estado :
O O X 0 1 2
X X O 3 4 5
O X . 6 7 8

Jugador : HUMANO
Los movimientos permitidos son:
(QUIERES UBICAR-FICHA-EN 8) (Teclea el numero 0)
Tu turno: 1
1 es ilegal.
Estado :
O O X 0 1 2
X X O 3 4 5
O X . 6 7 8

Jugador : HUMANO
Los movimientos permitidos son:
(QUIERES UBICAR-FICHA-EN 8) (Teclea el numero 0)
Tu turno: 1
1 es ilegal.

Estado :
O O X 0 1 2
X X O 3 4 5
O X . 6 7 8

Jugador : HUMANO
Los movimientos permitidos son:
(QUIERES UBICAR-FICHA-EN 8) (Teclea el numero 0)
Tu turno: 0

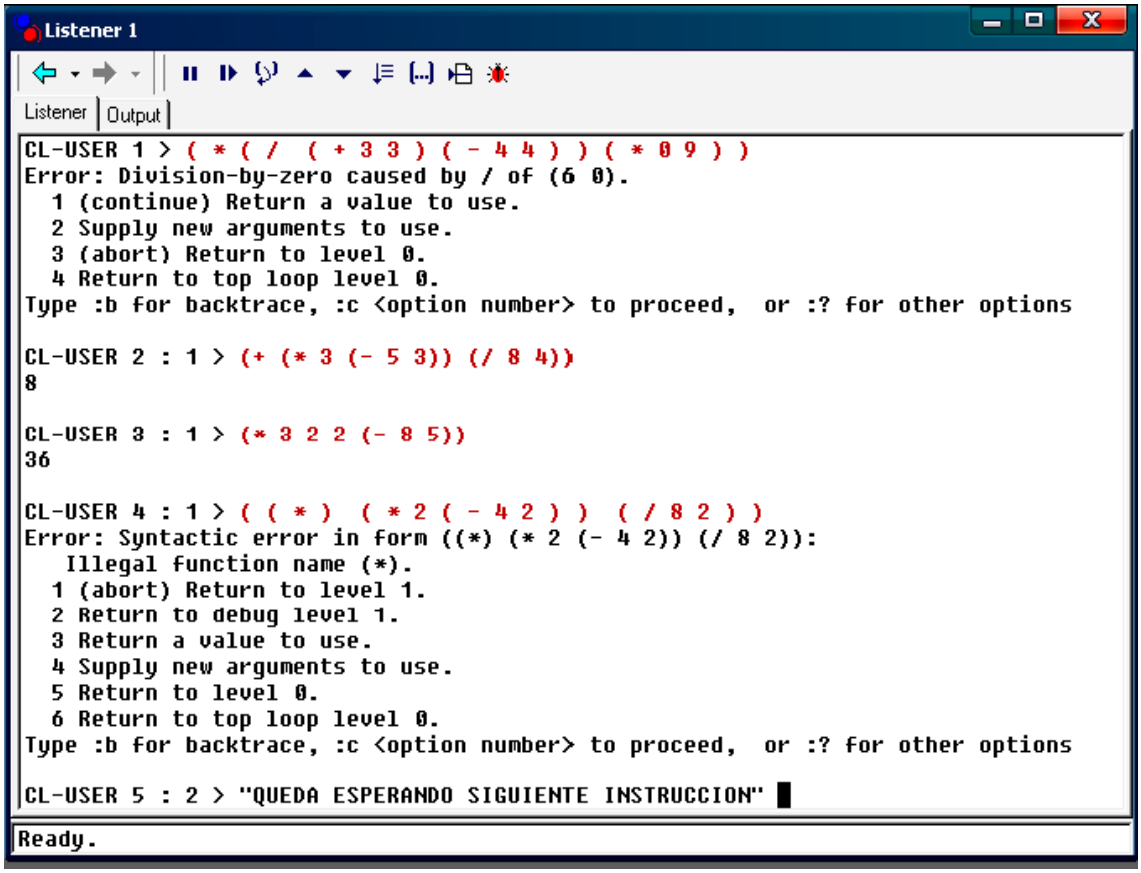
Estado :
O O X 0 1 2
X X O 3 4 5
O X X 6 7 8

Jugador : ORDENADOR
Empate
NIL

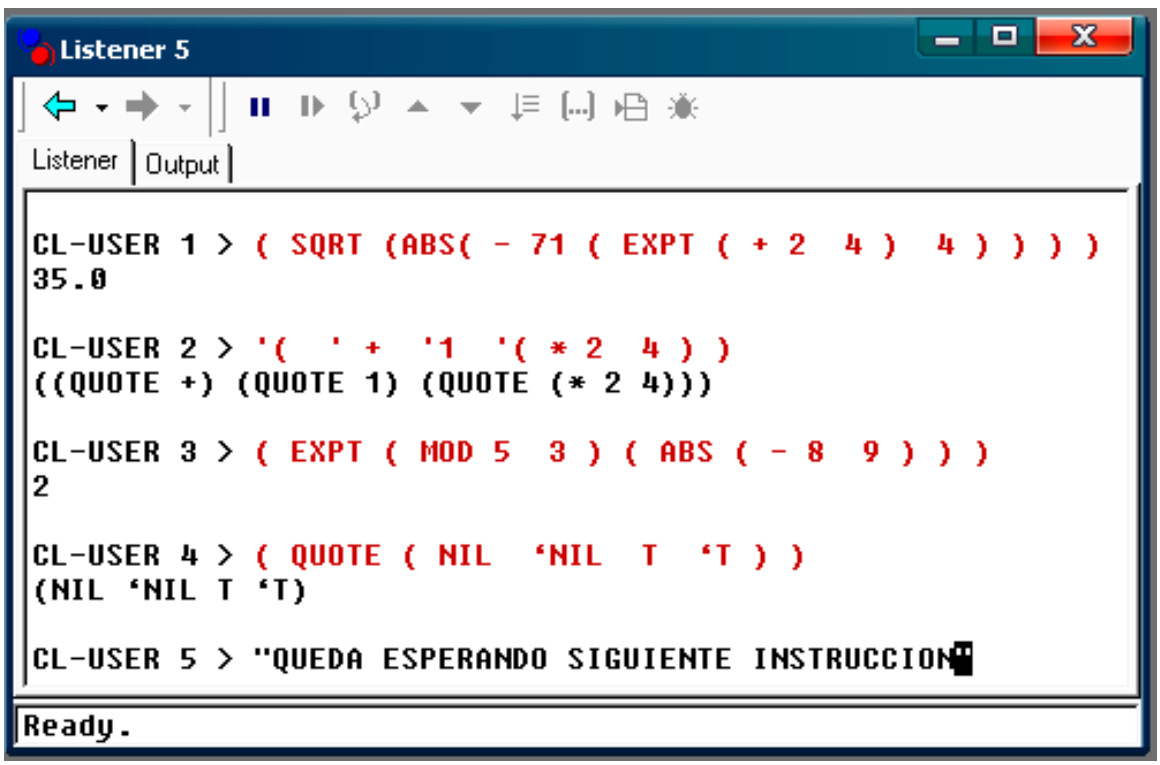
SOLUCIONES

SOLUCIÓN PRACTICA 2

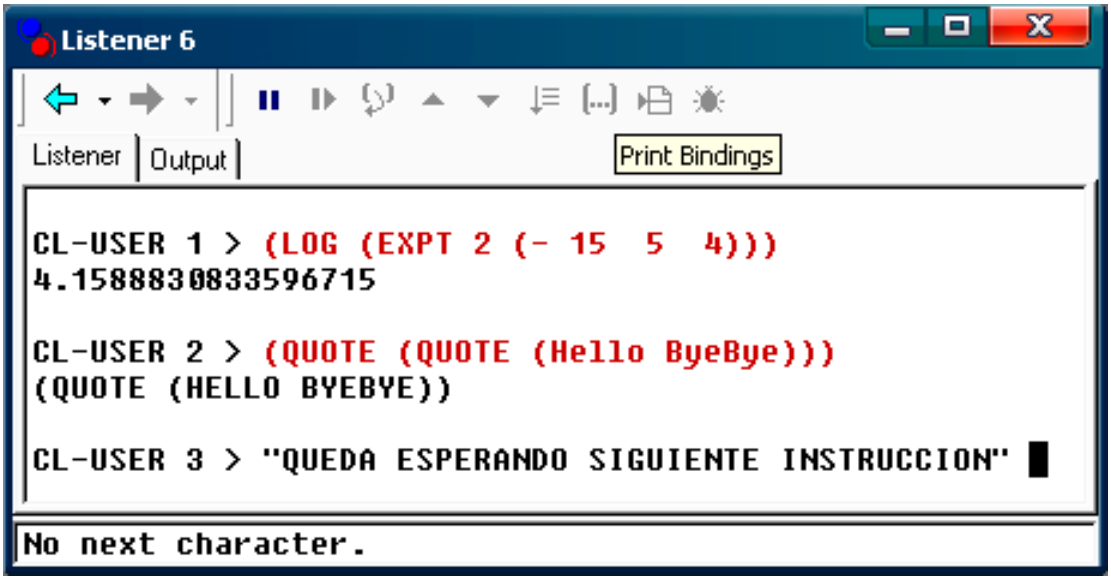
Solución 2-1 a) b) c) d).



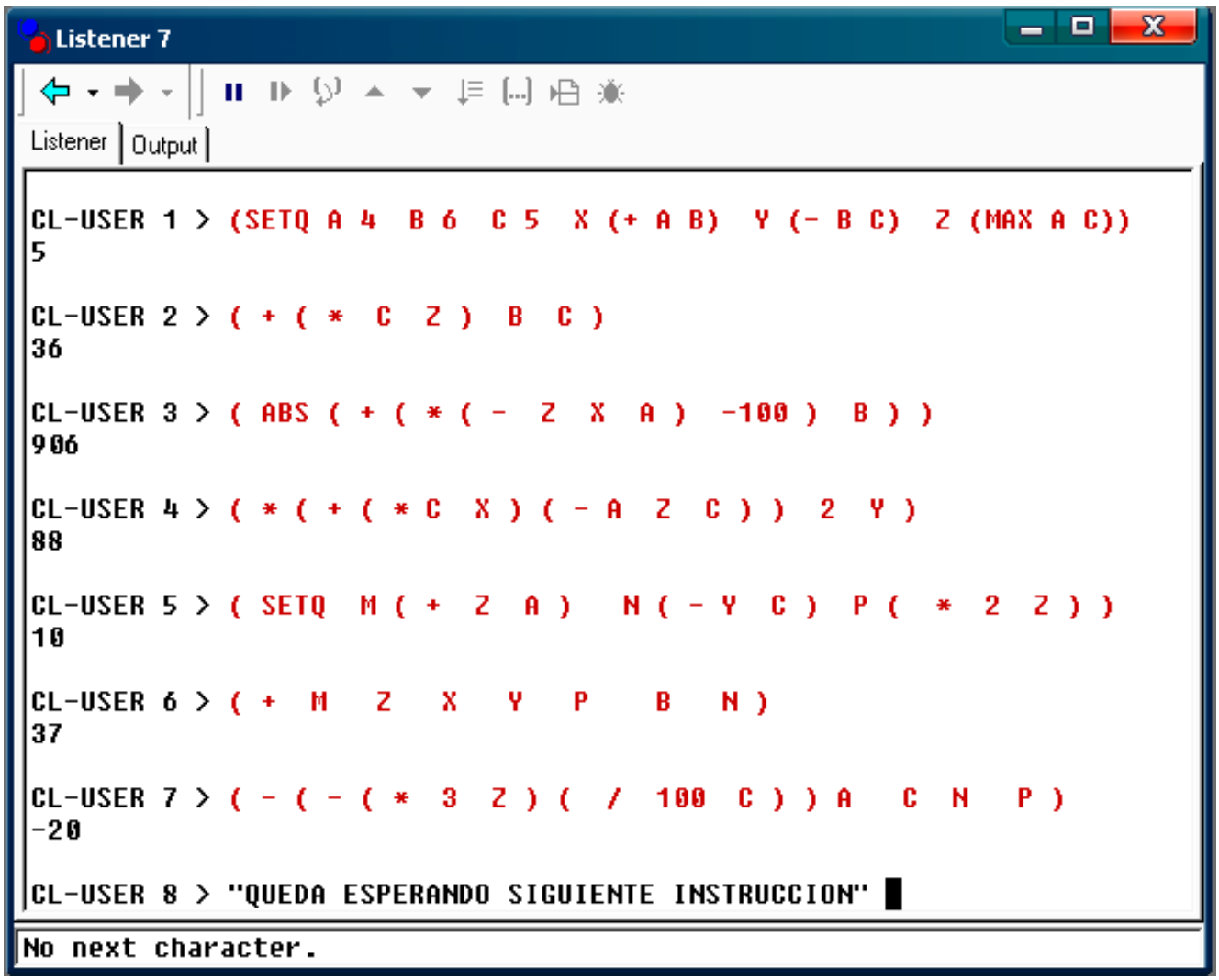
Solución 2-2 a) b) c) d).



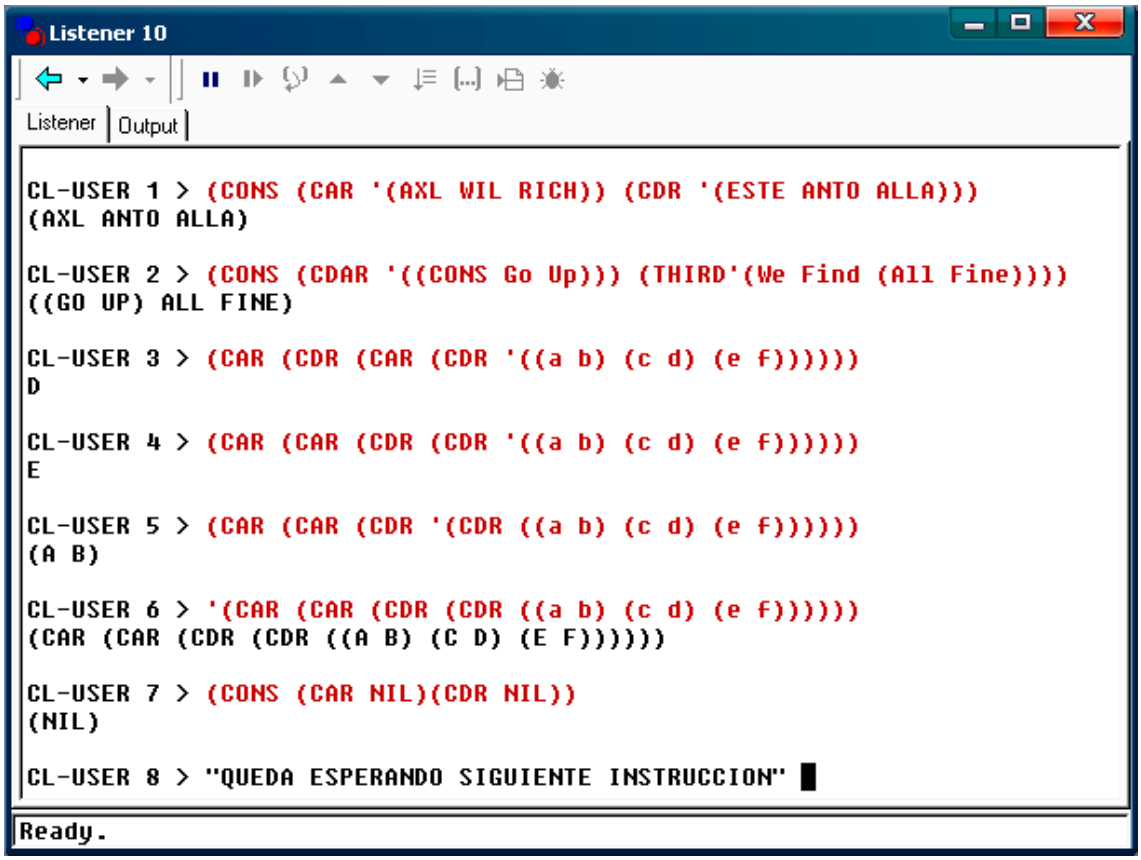
e) f) .



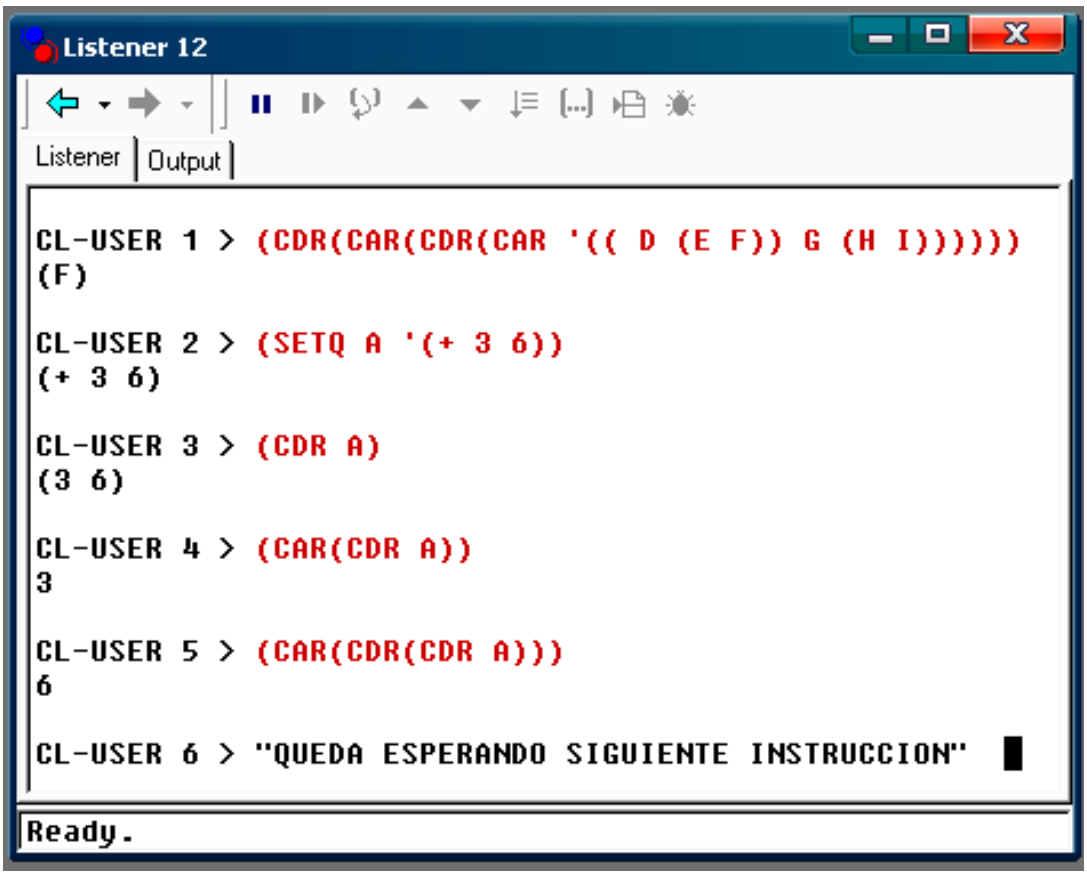
Solución 2-3 a) b) c) d) e).



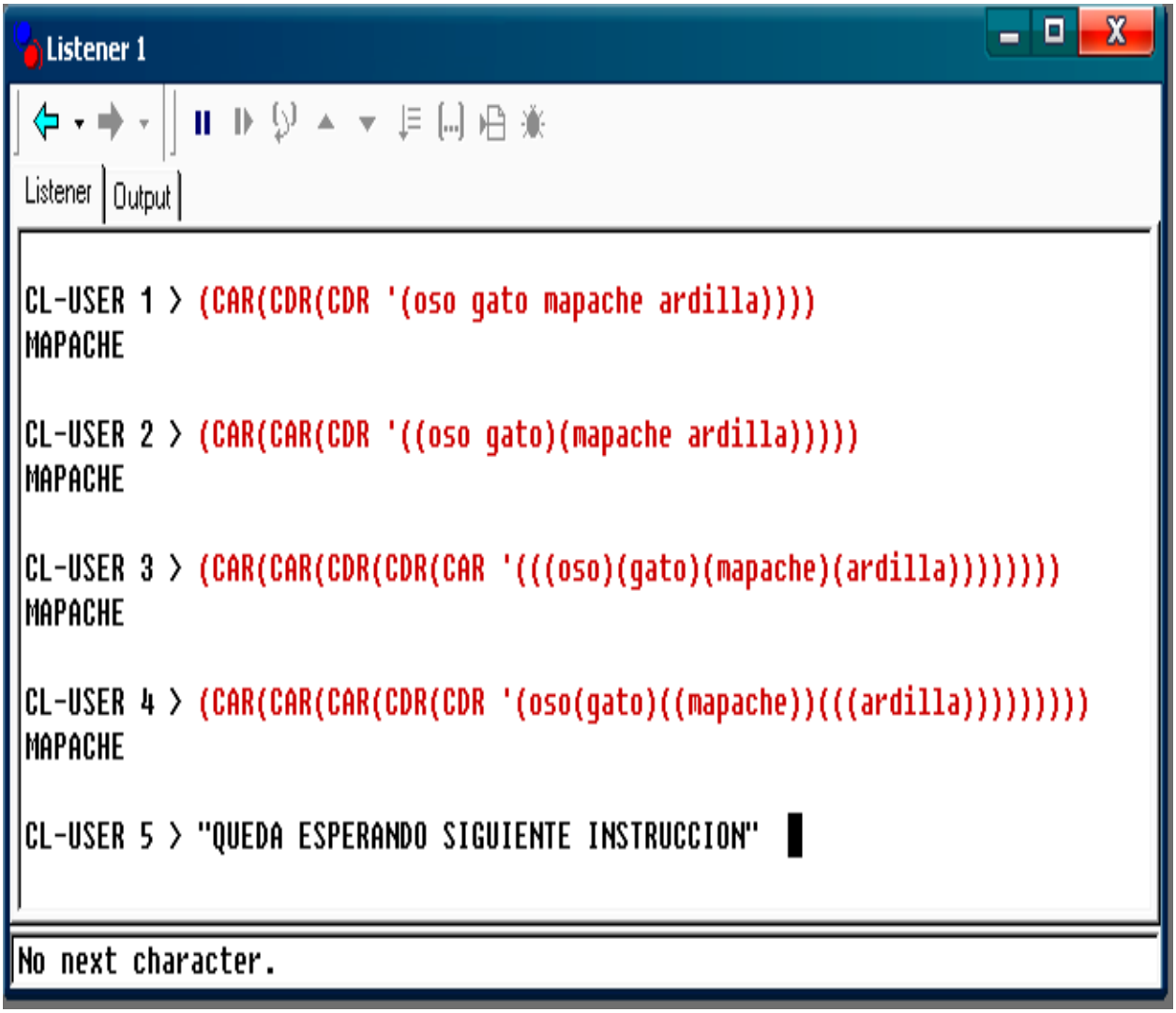
Solución 2-4 a) b) c) d) e) f) g)



Solución 2-5 a) b) b.1) b.2) b.3)



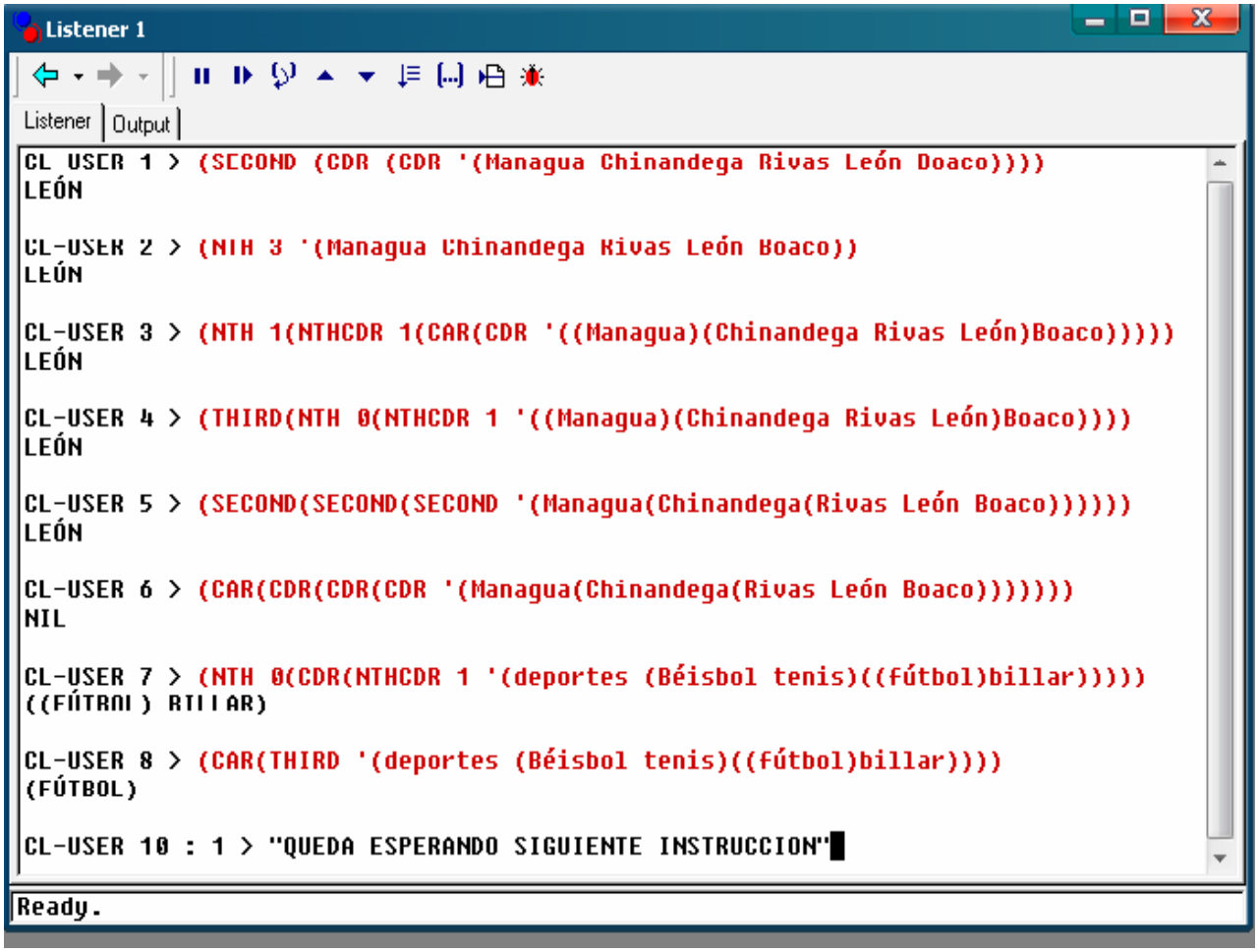
Solución 2-6 a) b) c) d)



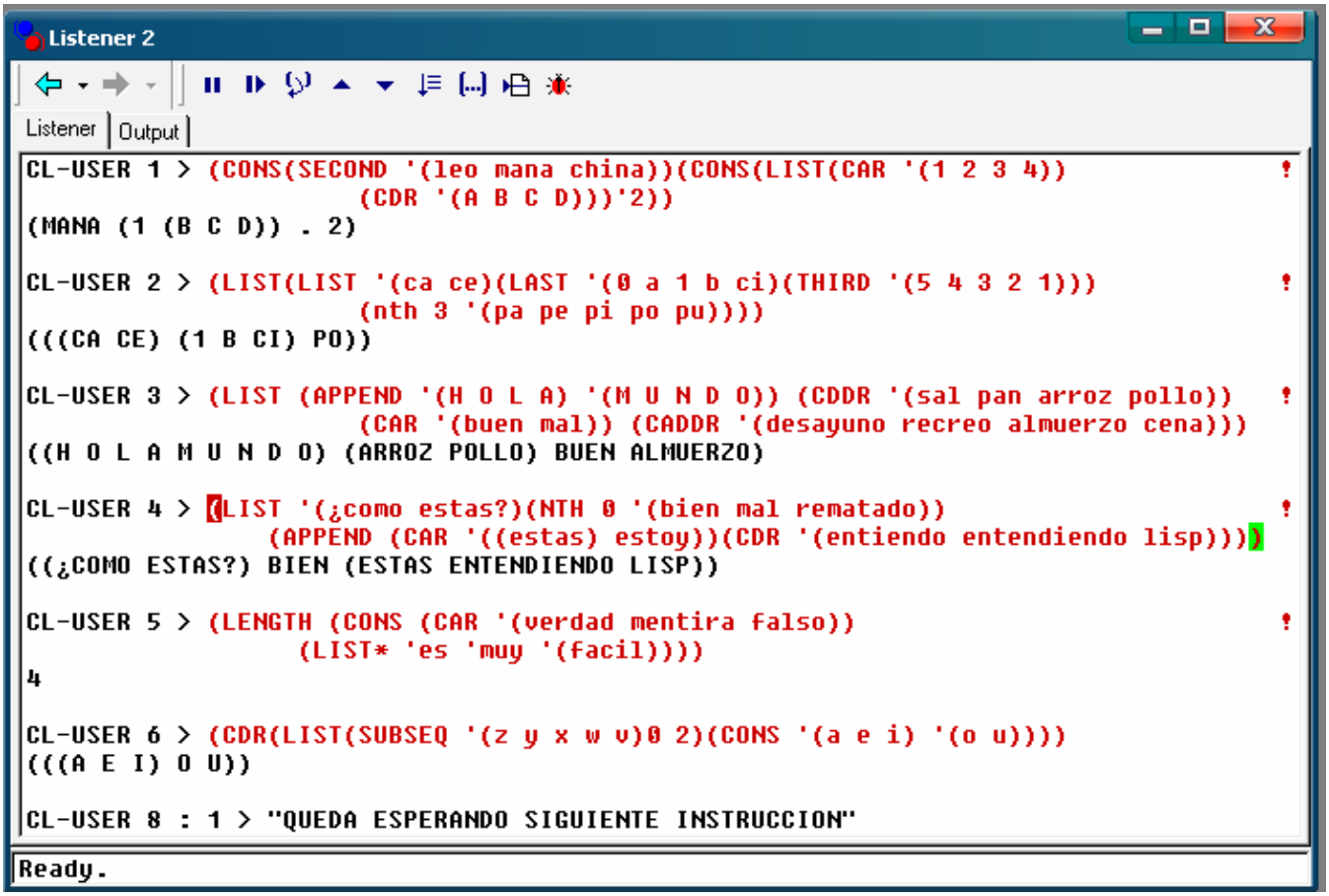
SOLUCIÓN PRÁCTICA 3

Solución 3-1 a) b) c) d)

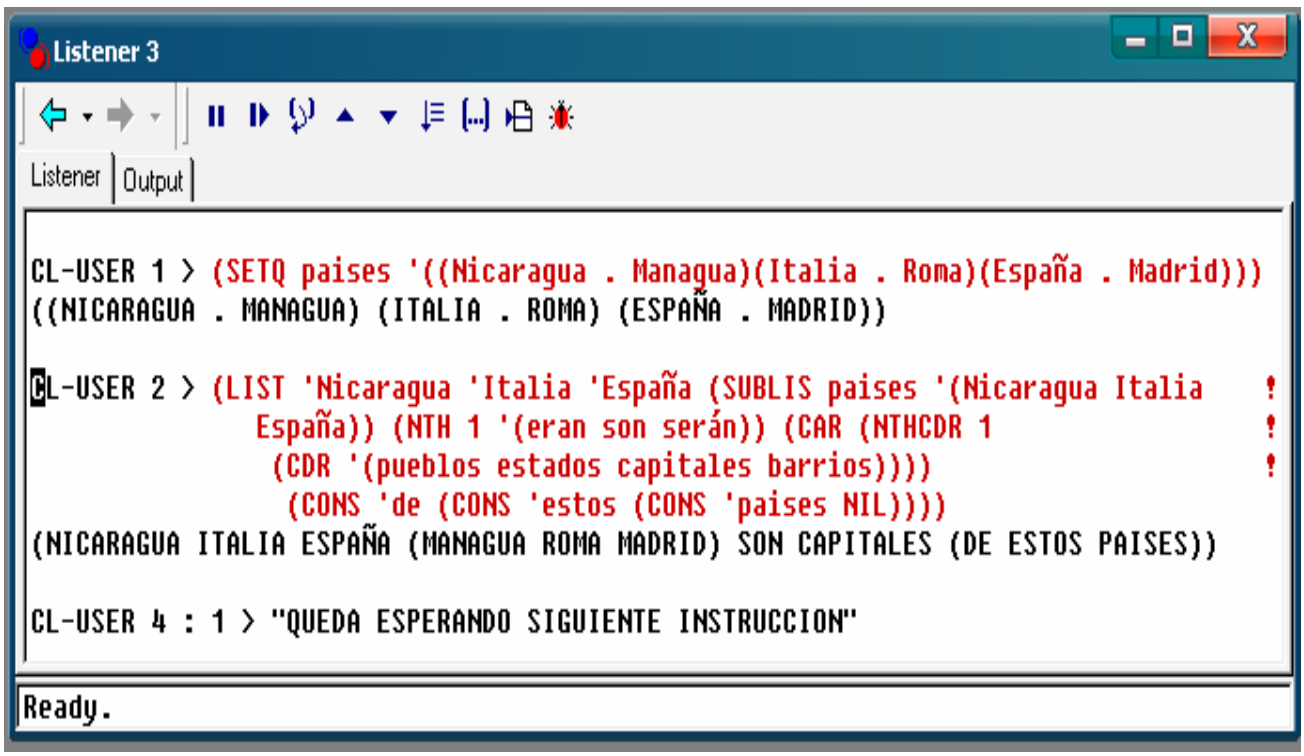
Nota: Se le darán 2 soluciones por cada ejercicio.



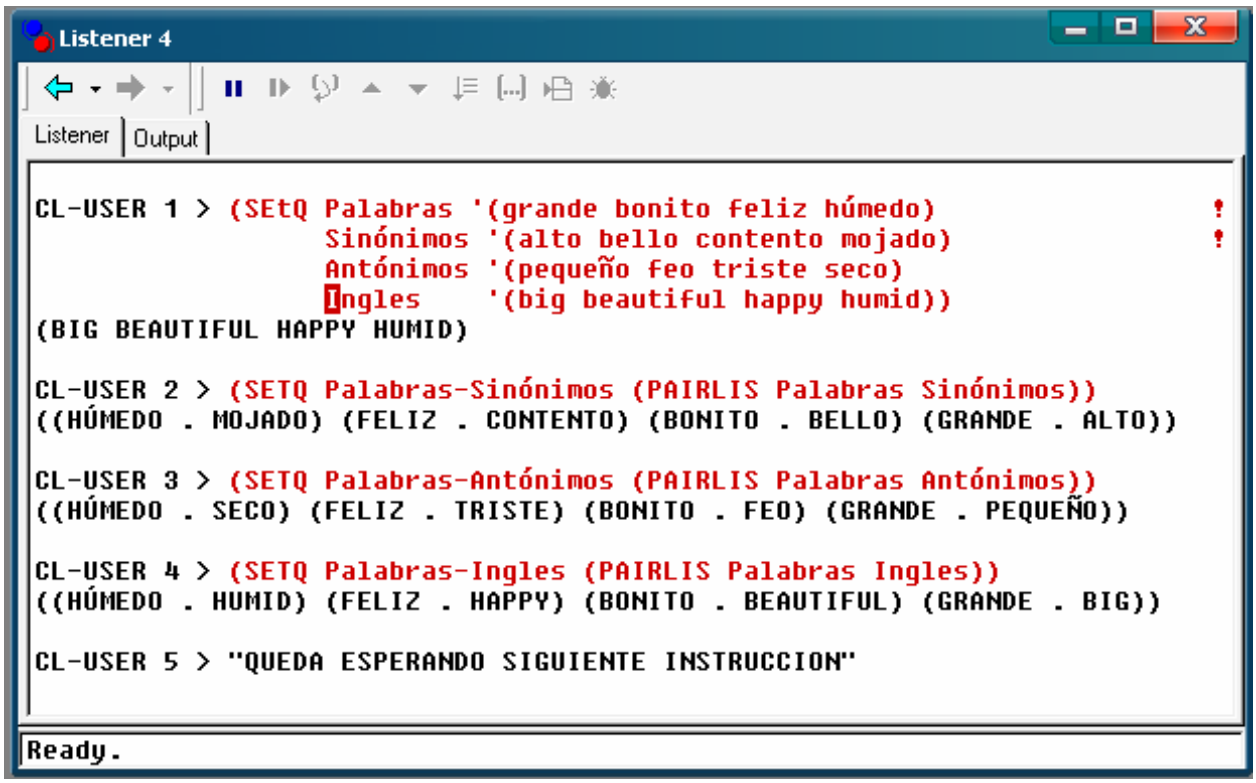
Solución 3-2 a) b) c) d) e) f)



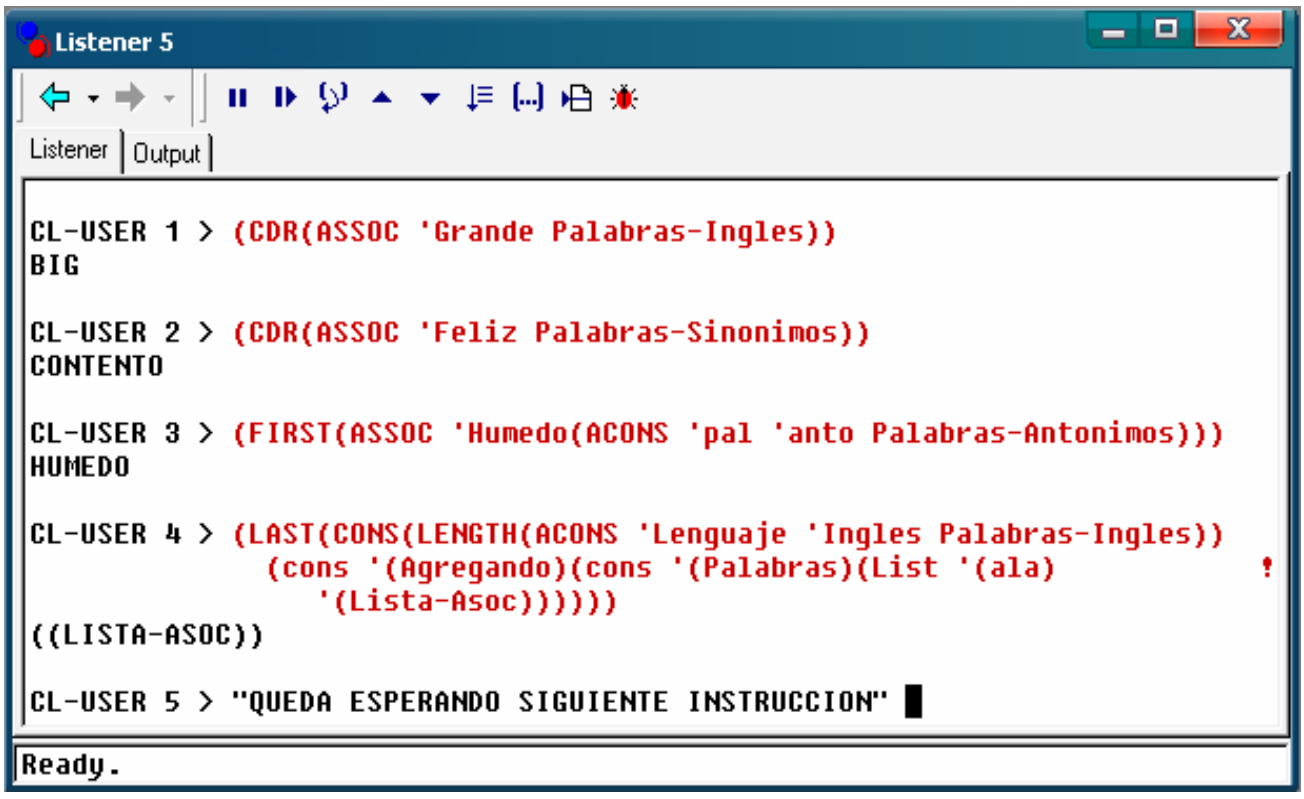
Solucion 3-3



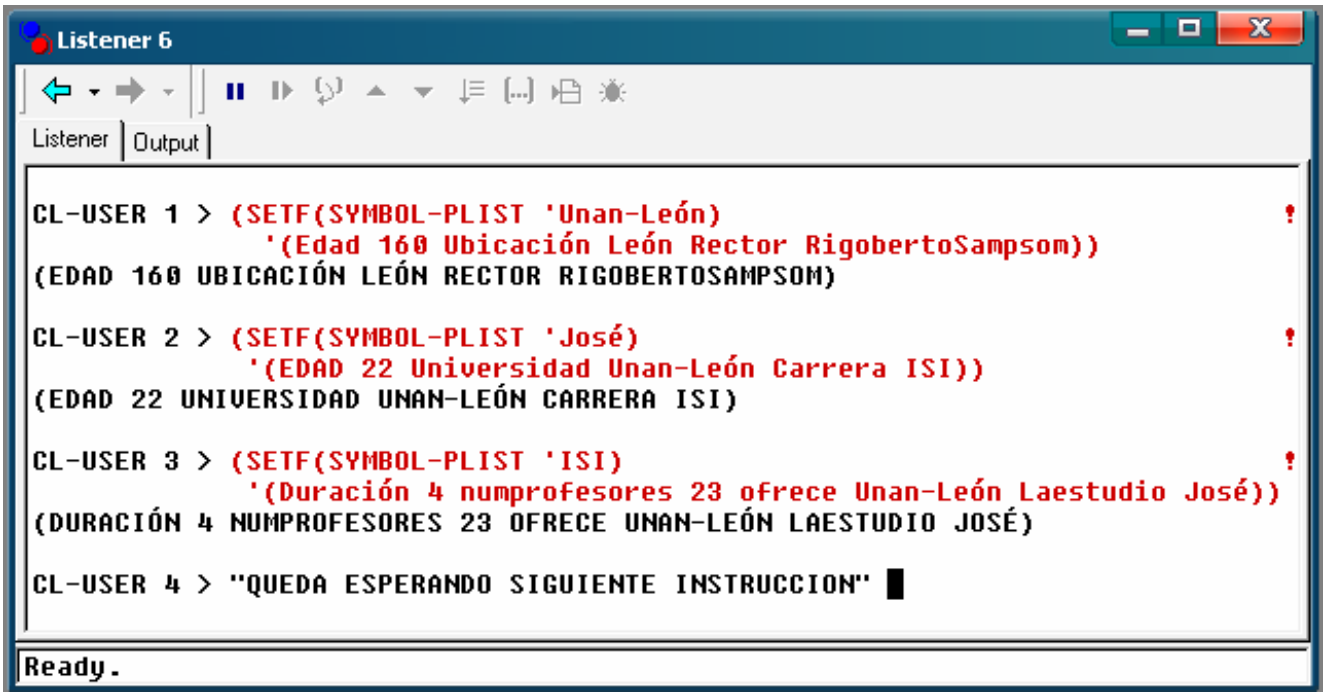
Solución 3-4 a)



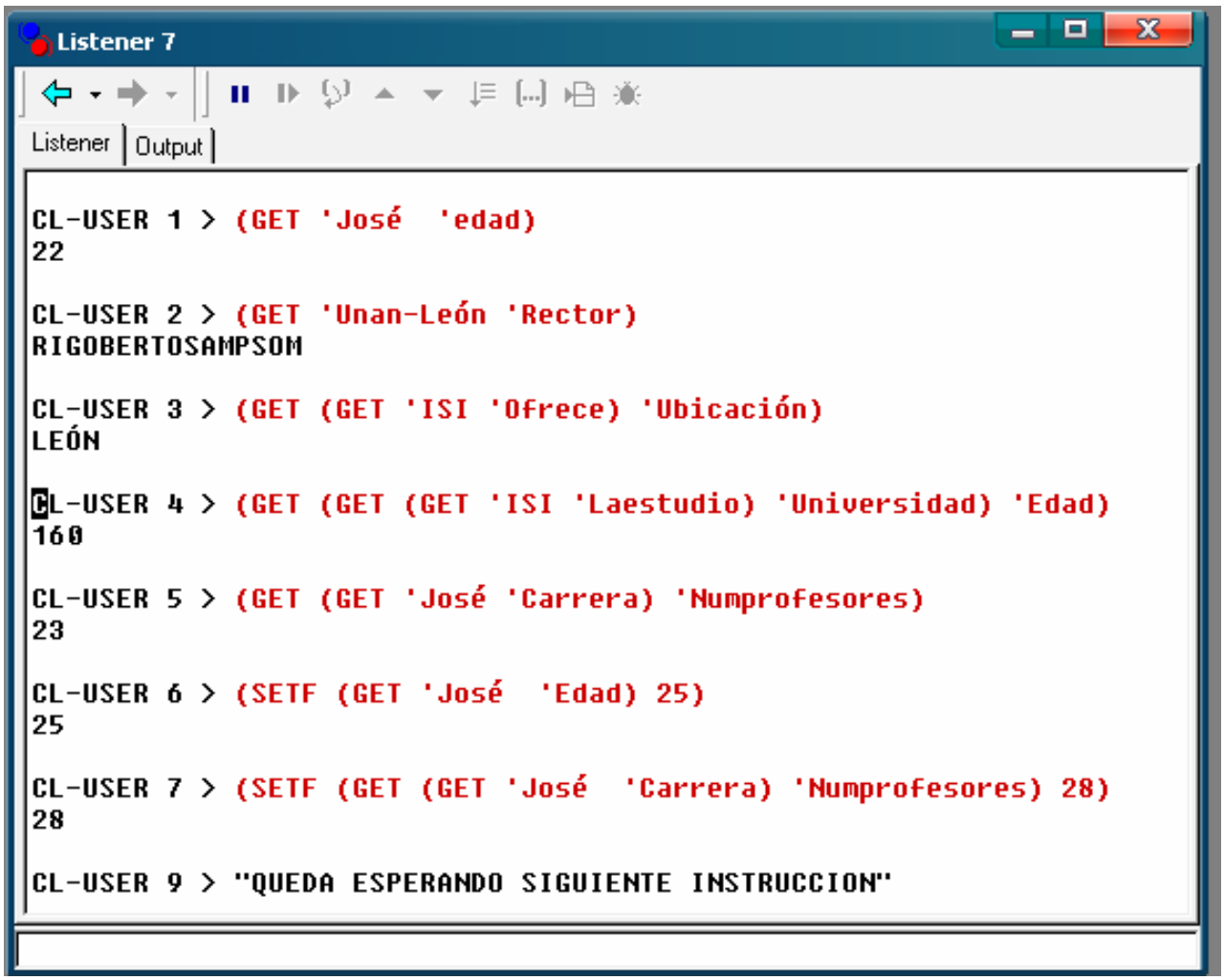
b) i. ii. iii. iv.



Solución 3-5

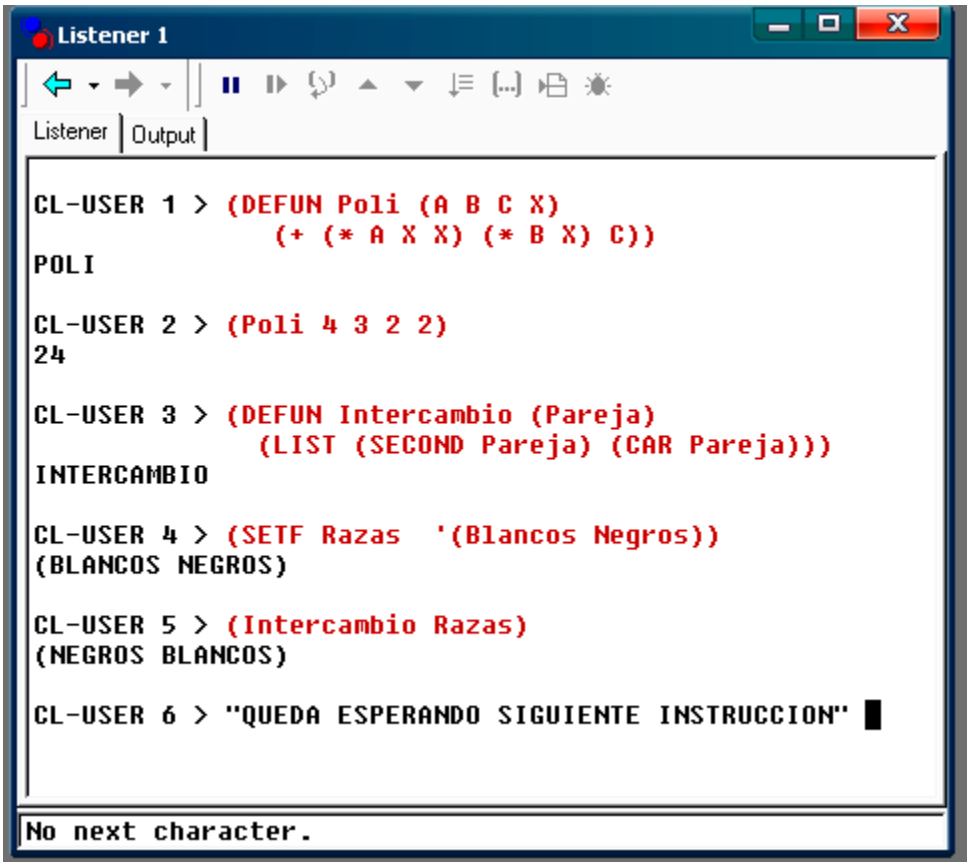


Solución 3-5.1

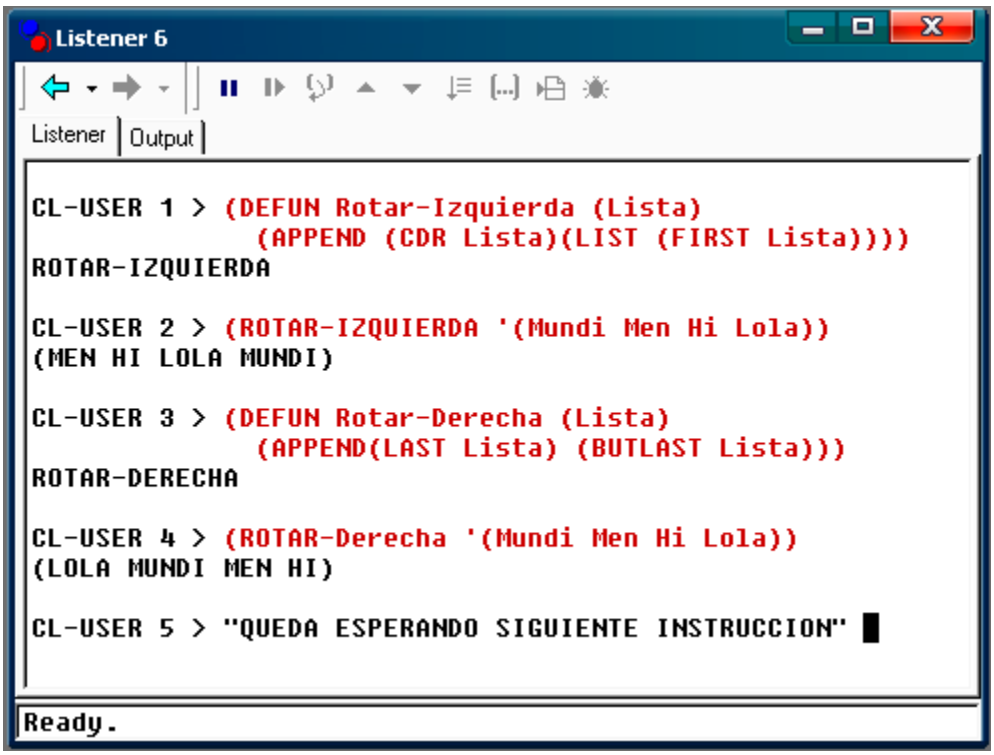


SOLUCIÓN PRÁCTICA 4

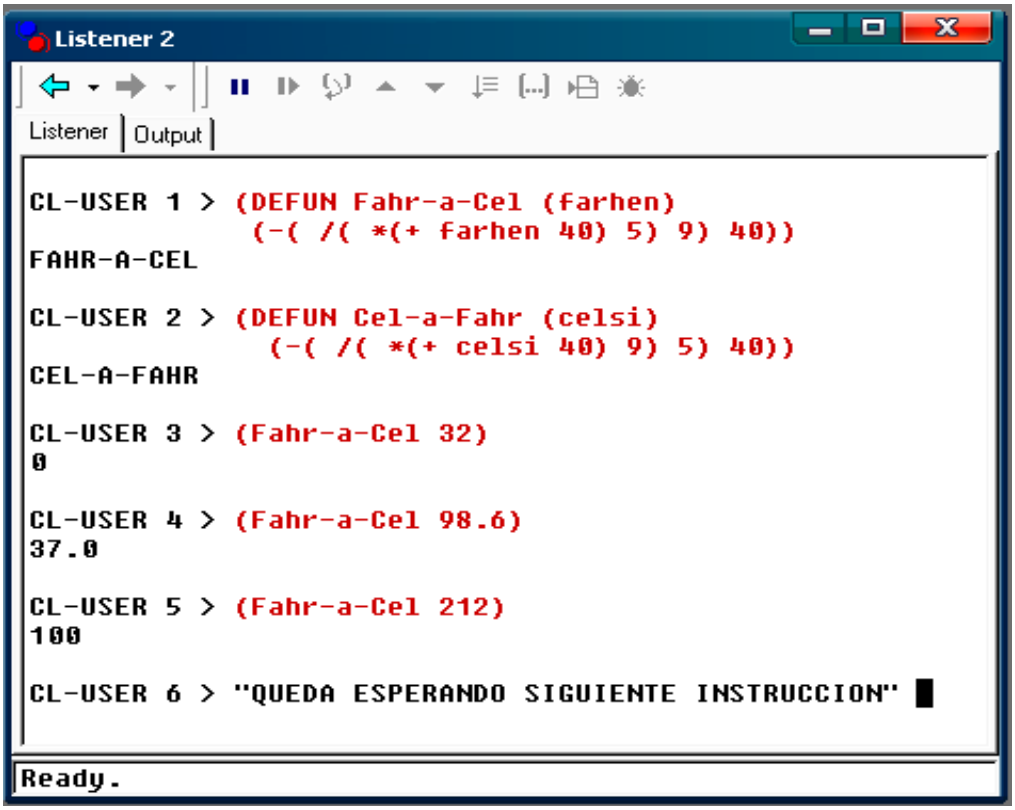
Solución 4-1 a) b)



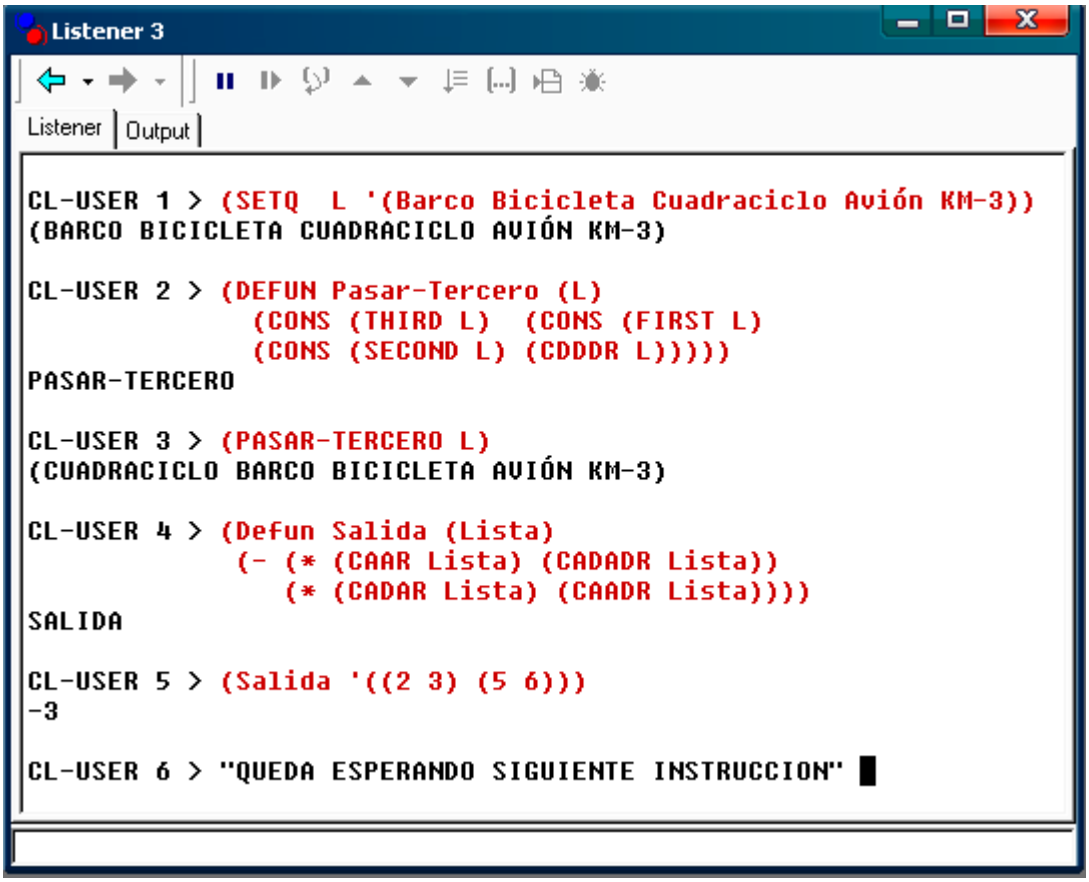
c) d).



e)

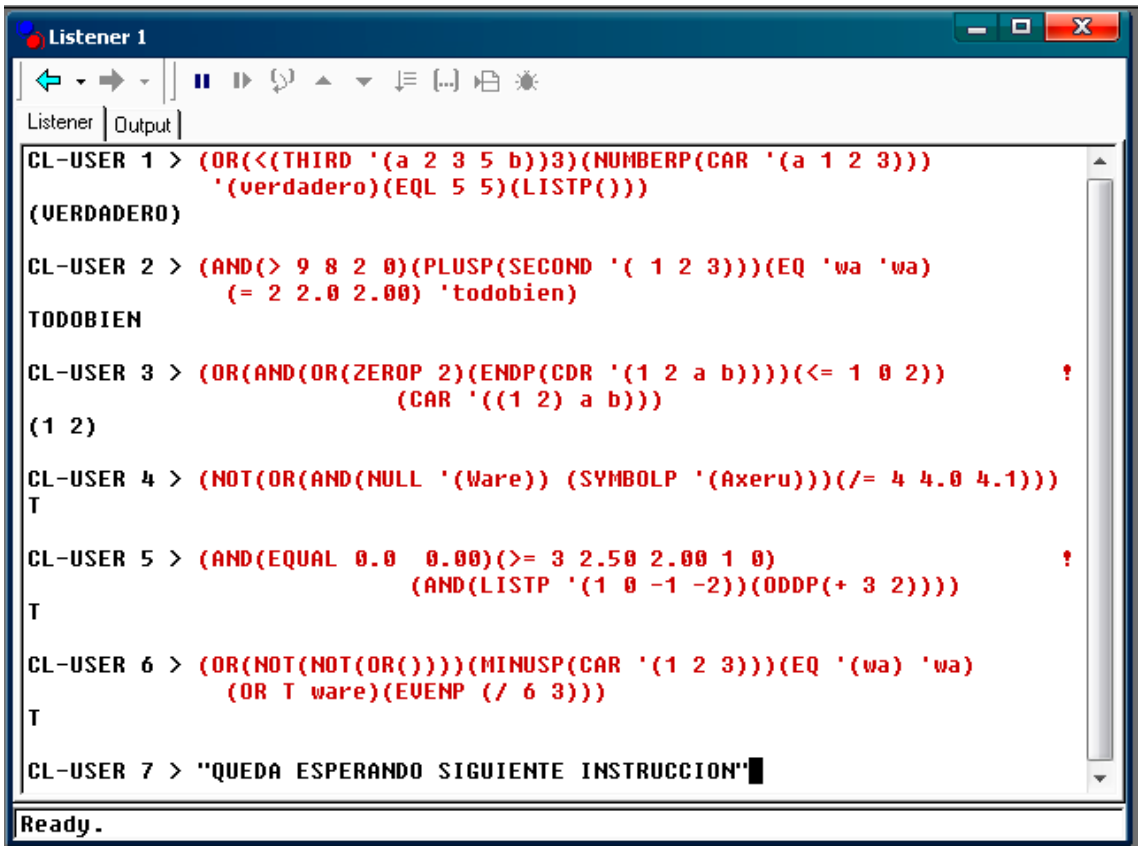

Listener 2
Listener
Output
CL-USER 1 > (DEFUN Fahr-a-Cel (farhen)
(-(/(*(+ farhen 40) 5) 9) 40))
FAHR-A-CEL
CL-USER 2 > (DEFUN Cel-a-Fahr (celsi)
(-(/(*(+ celsi 40) 9) 5) 40))
CEL-A-FAHR
CL-USER 3 > (Fahr-a-Cel 32)
0
CL-USER 4 > (Fahr-a-Cel 98.6)
37.0
CL-USER 5 > (Fahr-a-Cel 212)
100
CL-USER 6 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
Ready.

f) g)

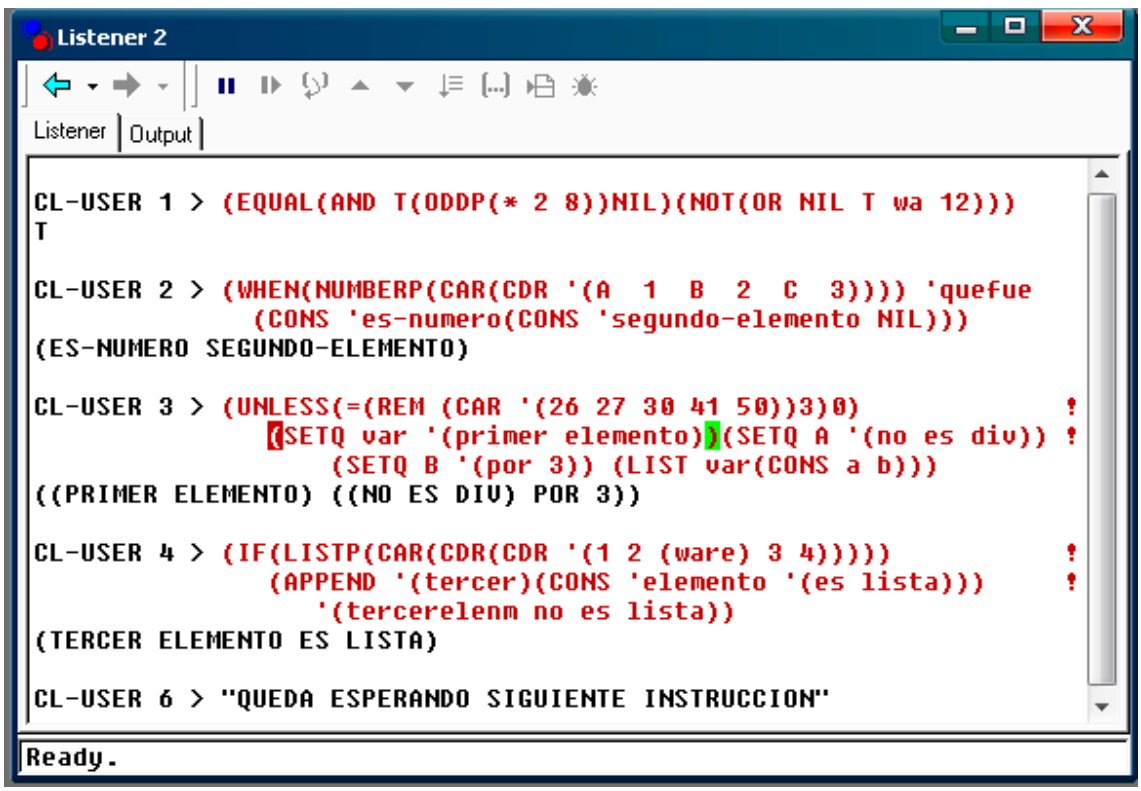

Listener 3
Listener
Output
CL-USER 1 > (SETQ L '(Barco Bicicleta Cuadraciclo Avión KM-3))
(BARCO BICICLETA CUADRACICLO AVIÓN KM-3)
CL-USER 2 > (DEFUN Pasar-Tercero (L)
(CONS (THIRD L) (CONS (FIRST L)
(CONS (SECOND L) (CDDDR L)))))
PASAR-TERCERO
CL-USER 3 > (PASAR-TERCERO L)
(CUADRACICLO BARCO BICICLETA AVIÓN KM-3)
CL-USER 4 > (Defun Salida (Lista)
(- (* (CAAR Lista) (CADADR Lista))
(* (CADAR Lista) (CAADR Lista))))
SALIDA
CL-USER 5 > (Salida '((2 3) (5 6)))
-3
CL-USER 6 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"

SOLUCIÓN PRÁCTICA 5

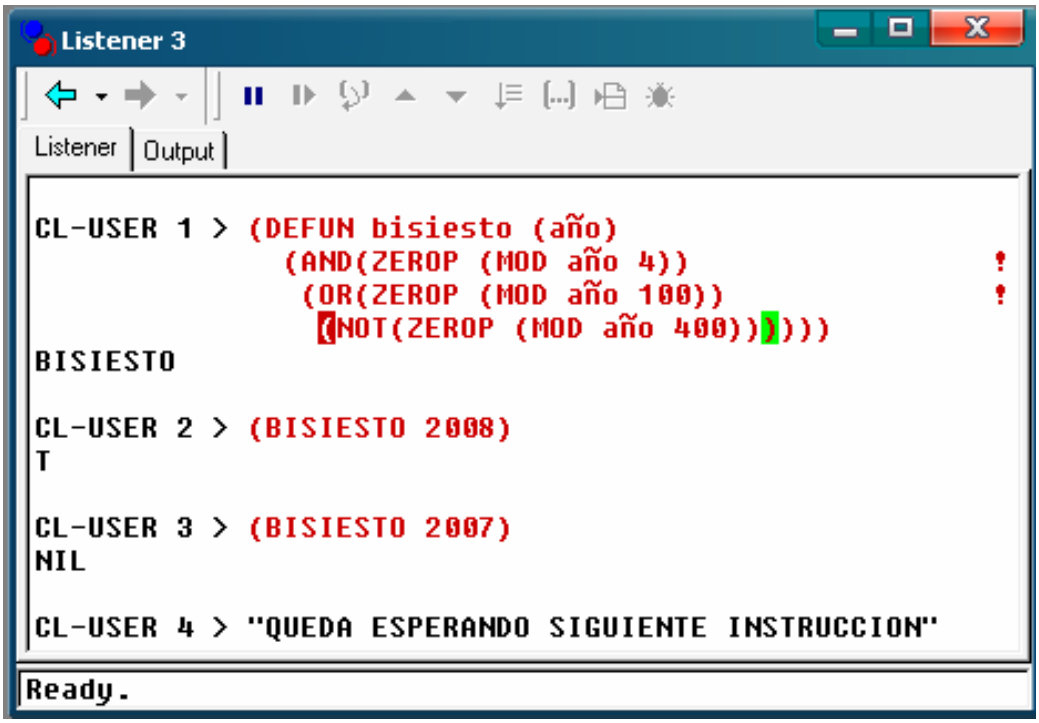
Solución 5-1. a) b) c) d) e) f)



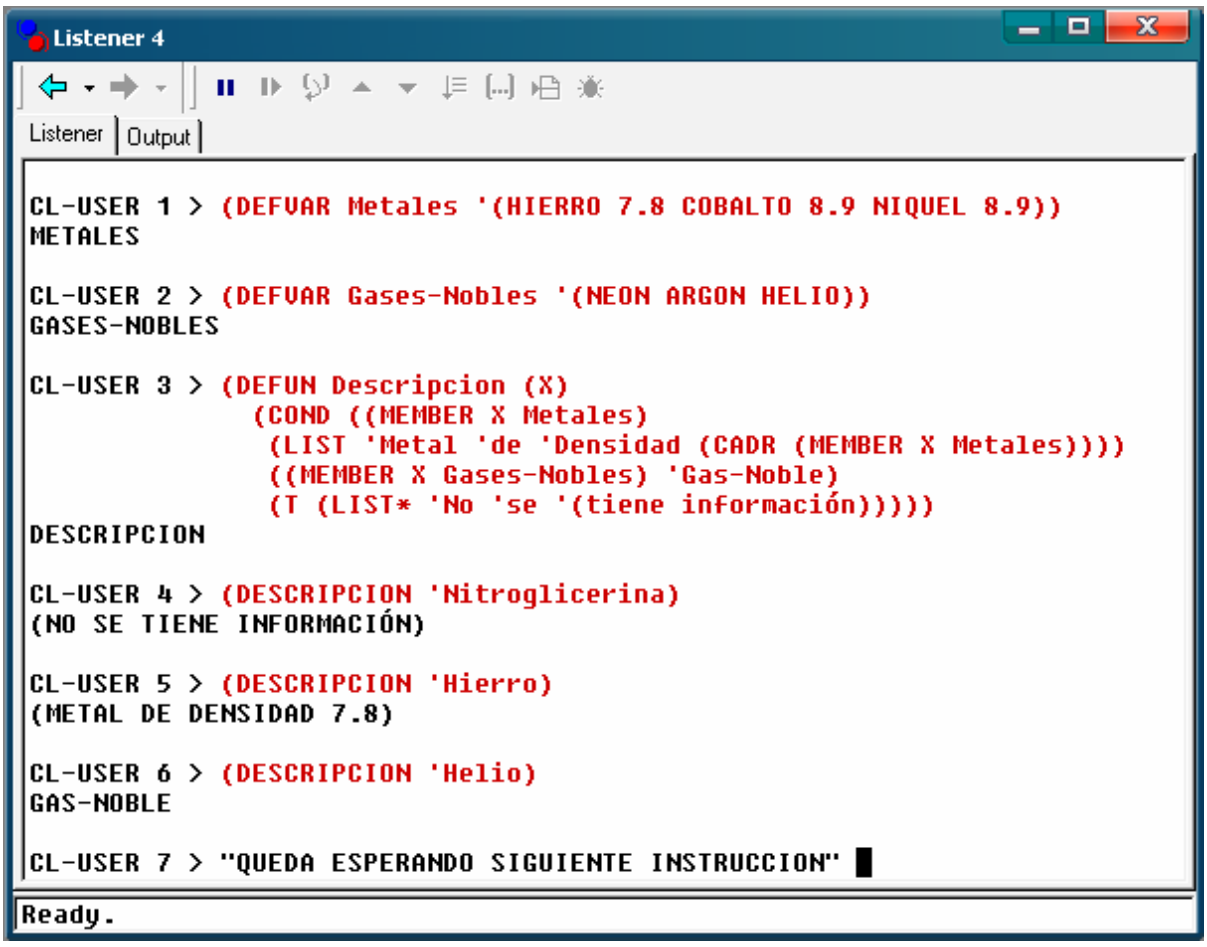
g) h) i) j)



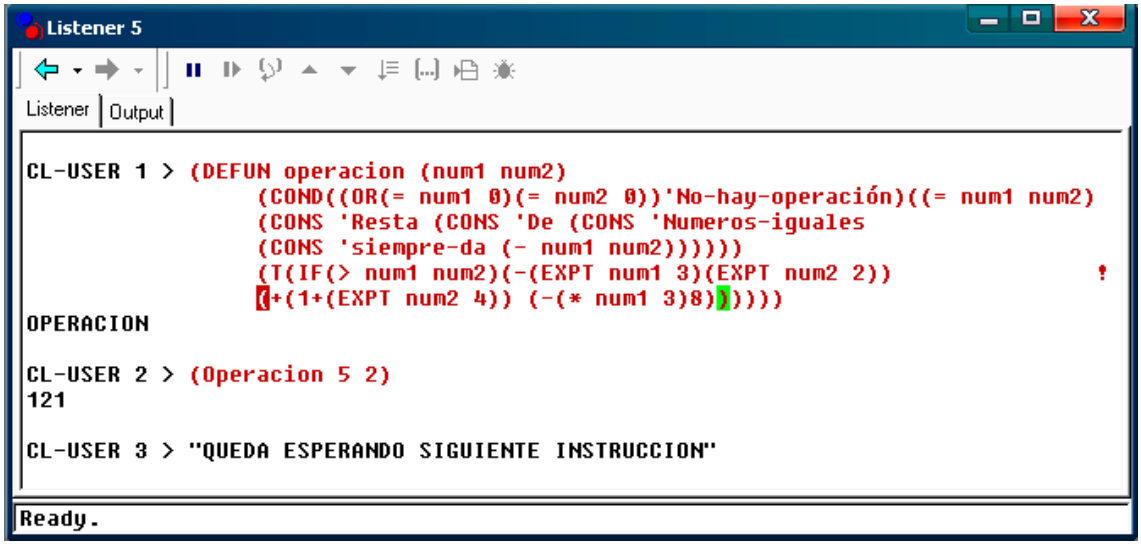
Solución 5-2 A.



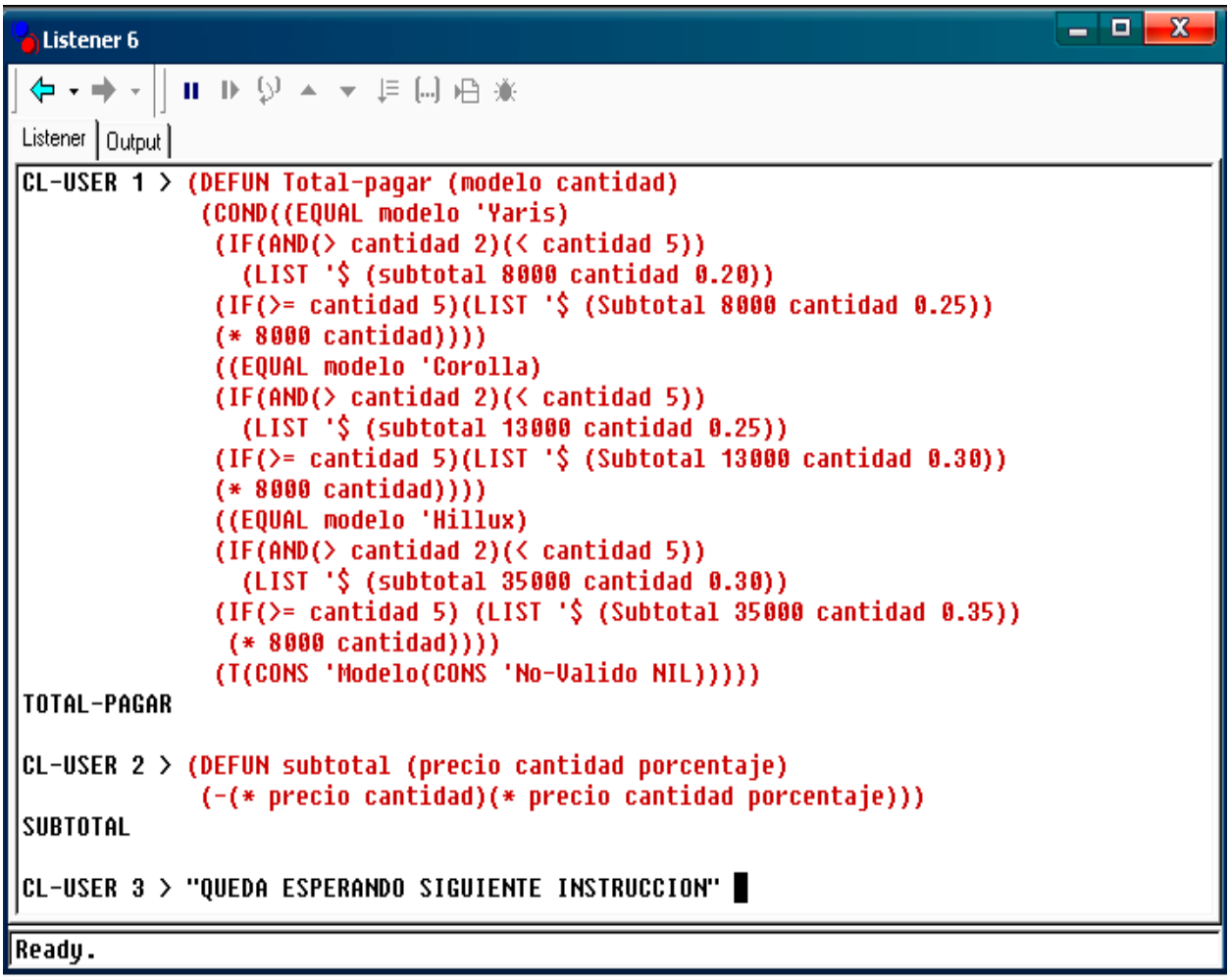
B.

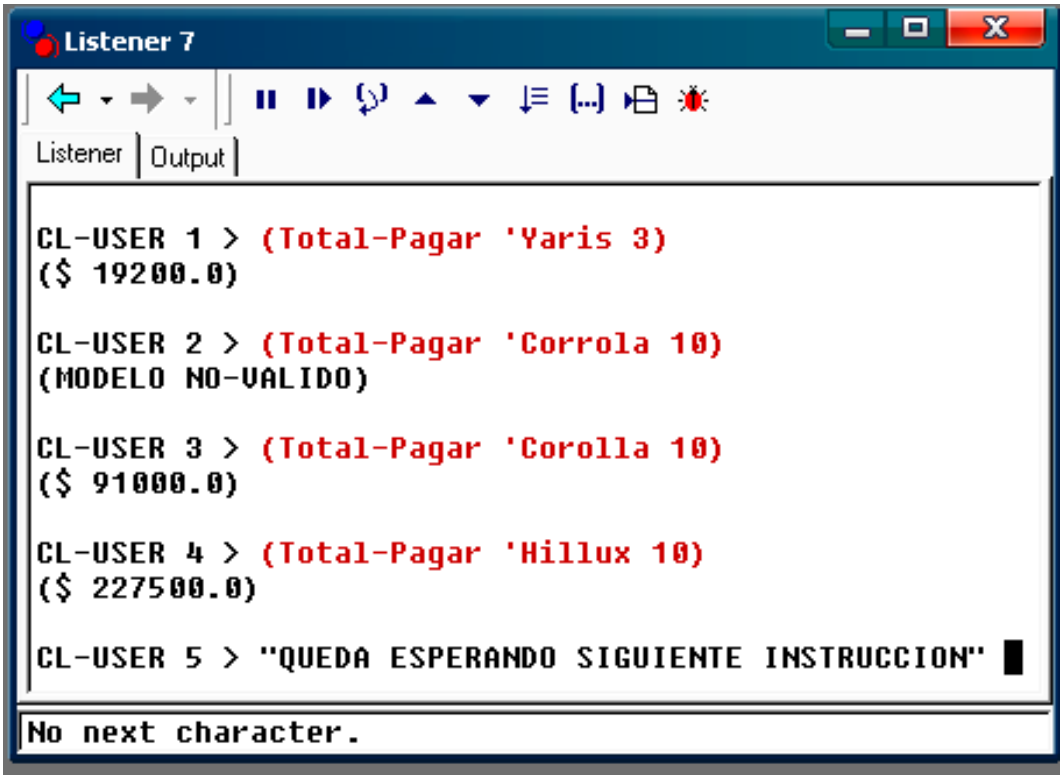


C.

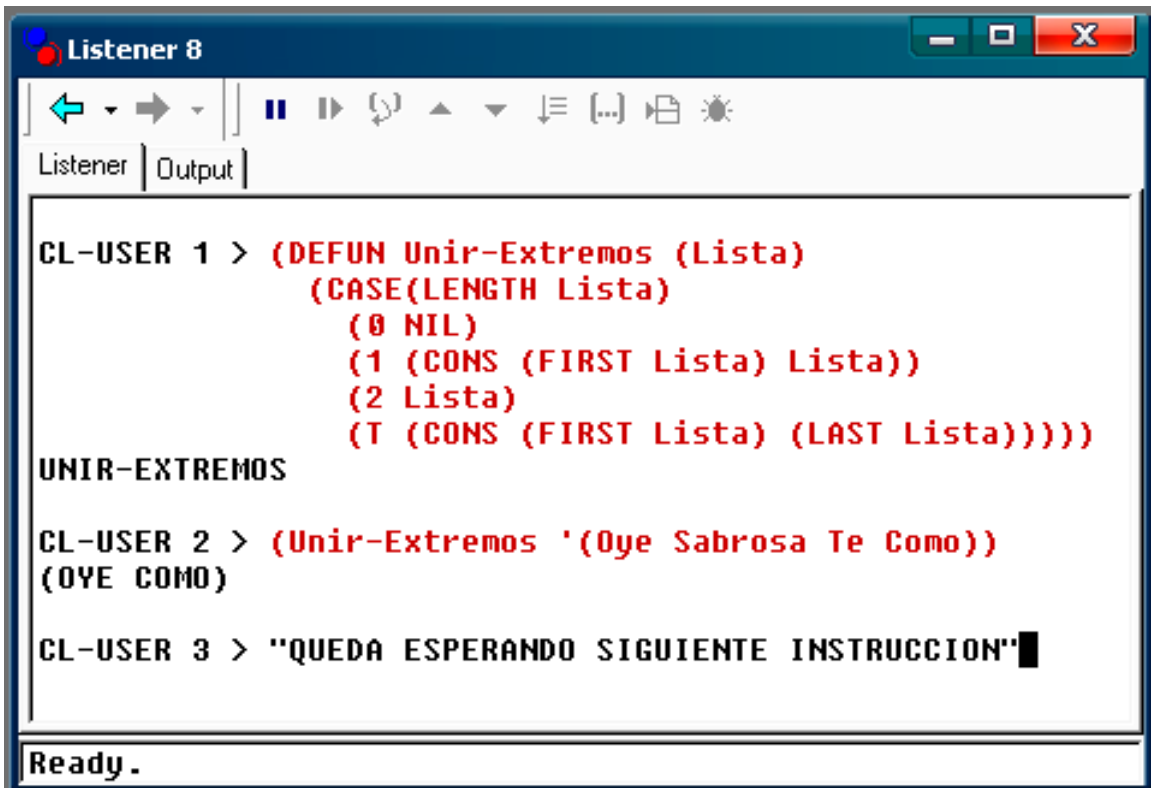

Listener 5
Listener
Output
CL-USER 1 > (DEFUN operacion (num1 num2)
(COND((OR(= num1 0)(= num2 0))'No-hay-operación)((= num1 num2)
(CONS 'Resta (CONS 'De (CONS 'Numeros-iguales
(CONS 'siempre-da (- num1 num2))))))
(T(IF(> num1 num2)(-(EXPT num1 3)(EXPT num2 2))
(+(1+(EXPT num2 4)) (-(* num1 3)8)))))
OPERACION
CL-USER 2 > (Operacion 5 2)
121
CL-USER 3 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
Ready.

D.


Listener 6
Listener
Output
CL-USER 1 > (DEFUN Total-pagar (modelo cantidad)
(COND((EQUAL modelo 'Yaris)
(IF(AND(> cantidad 2)(< cantidad 5))
(LIST '$ (subtotal 8000 cantidad 0.20))
(IF(>= cantidad 5)(LIST '$ (Subtotal 8000 cantidad 0.25))
(* 8000 cantidad))))
((EQUAL modelo 'Corolla)
(IF(AND(> cantidad 2)(< cantidad 5))
(LIST '$ (subtotal 13000 cantidad 0.25))
(IF(>= cantidad 5)(LIST '$ (Subtotal 13000 cantidad 0.30))
(* 8000 cantidad))))
((EQUAL modelo 'Hillux)
(IF(AND(> cantidad 2)(< cantidad 5))
(LIST '$ (subtotal 35000 cantidad 0.30))
(IF(>= cantidad 5) (LIST '$ (Subtotal 35000 cantidad 0.35))
(* 8000 cantidad))))
(T(CONS 'Modelo(CONS 'No-Valido NIL)))))
TOTAL-PAGAR
CL-USER 2 > (DEFUN subtotal (precio cantidad porcentaje)
(-(* precio cantidad)(* precio cantidad porcentaje)))
SUBTOTAL
CL-USER 3 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
Ready.


Listener 7
Listener
Output
CL-USER 1 > (Total-Pagar 'Yaris 3)
($ 19200.0)
CL-USER 2 > (Total-Pagar 'Corrola 10)
(MODELO NO-VALIDO)
CL-USER 3 > (Total-Pagar 'Corolla 10)
($ 91000.0)
CL-USER 4 > (Total-Pagar 'Hillux 10)
($ 227500.0)
CL-USER 5 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
No next character.

E.


Listener 8
Listener
Output
CL-USER 1 > (DEFUN Unir-Extremos (Lista)
(CASE(LENGTH Lista)
(0 NIL)
(1 (CONS (FIRST Lista) Lista))
(2 Lista)
(T (CONS (FIRST Lista) (LAST Lista)))))
UNIR-EXTREMOS
CL-USER 2 > (Unir-Extremos '(Oye Sabrosa Te Como))
(OYE COMO)
CL-USER 3 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
Ready.

SOLUCIÓN PRÁCTICA 6

Solución 6-1

a)

Las variables que encontramos en el entorno son X, Y, Z. Su respectiva inicialización es: X→(A B C), Y→(D E F G H) y luego Z→(A B). El siguiente paso es el CDR: X→(B C), Y→(E F G H), Z→(A B). Se harán otros CDR hasta que X cumpla con (NULL X) que es vacío. Al final del ultimo CDR quedaran: X→(), Y→(G H) y Z→(A B). Siendo esto verdadero se realiza el último paso (APPEND Y Z) que nos devuelve la respuesta: (G H A B).

b)

Las variables que encontramos en el entorno son X, Y, Z. Su respectiva inicialización es: X→(A B C), Y→(D E F G H) y Z→(). Se calcula (CONS X Y) quedando: ((A B C) D E F G H). Se prosigue con el CDR quedando: X→(B C), Y→(D E F G H) y Z→(). Otro (CONS X Y): ((B C) D E F G H). Otro CDR: X→(C), Y→(D E F G H) y Z→(). Otro (CONS X Y): ((C) D E F G H). Lo siguiente es el CDR: X→(), Y→(D E F G H) y Z→(). Ahora se satisface el (NULL X) y solo queda ejecutar el (CONS Y Z)) quedando como resultado: ((D E F G H)).

c)

Las variables locales que se encuentran son N y M. Su respectiva inicialización es N→1 y M→N. Las dos operaciones se llevan a cabo de manera secuencial y luego el valor de N esta ya disponible para M, o sea que M→1. En el siguiente paso los valores quedan: N→2 y M→2. El otro paso: N→3 y M→4. Otra vez el mismo procedimiento: N→4 y M→8. Una vez más lo mismo: N→5 y M→16. Ahora que la condición (> M 10) se satisface este retorna el valor de la última expresión siendo: 5.

d)

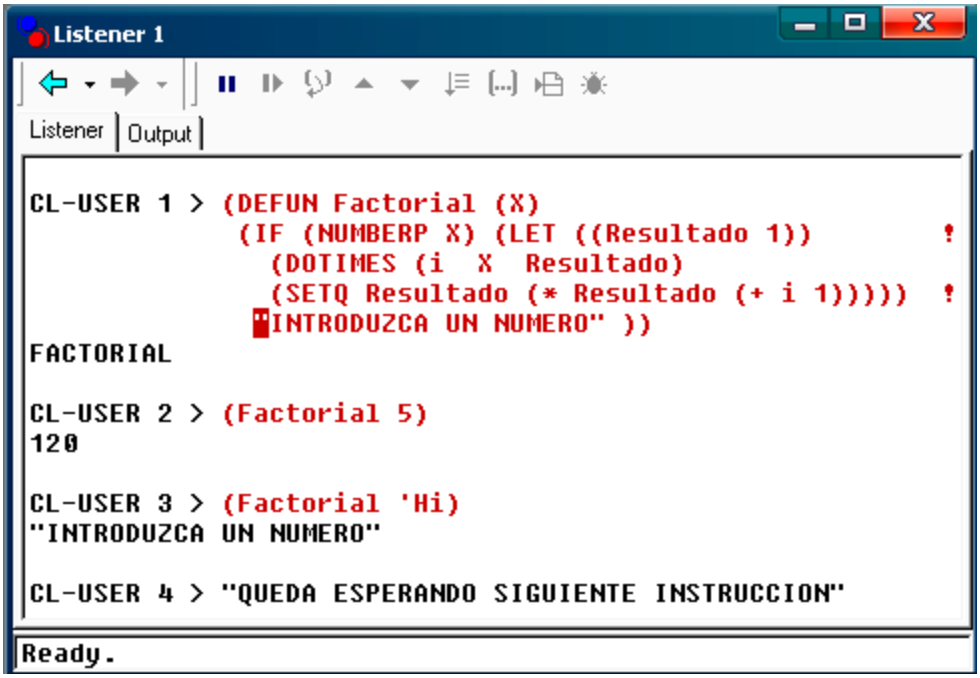
Supongamos que para la función “times” se le pasan los argumentos a→3 y b→4. El COND evaluara si ‘a’ es igual a 0, si lo es retornara el mismo 0 sino seguirá hasta encontrar T. Hace la suma b que es 4 y entra al “times” y resta 3 con 1 resultando 2 y manda a llamar a “times” de nuevo. De nuevo suma 4 y entra al “times” reduciendo 2 a 1 y la llama otra vez. Suma 4 de nuevo y reduce el 1 a 0 y llama a “times”. Al ser la condición a→0 se hace la suma de los tres primeros 4 que se tenían resultando 4+4+4=12.

e)

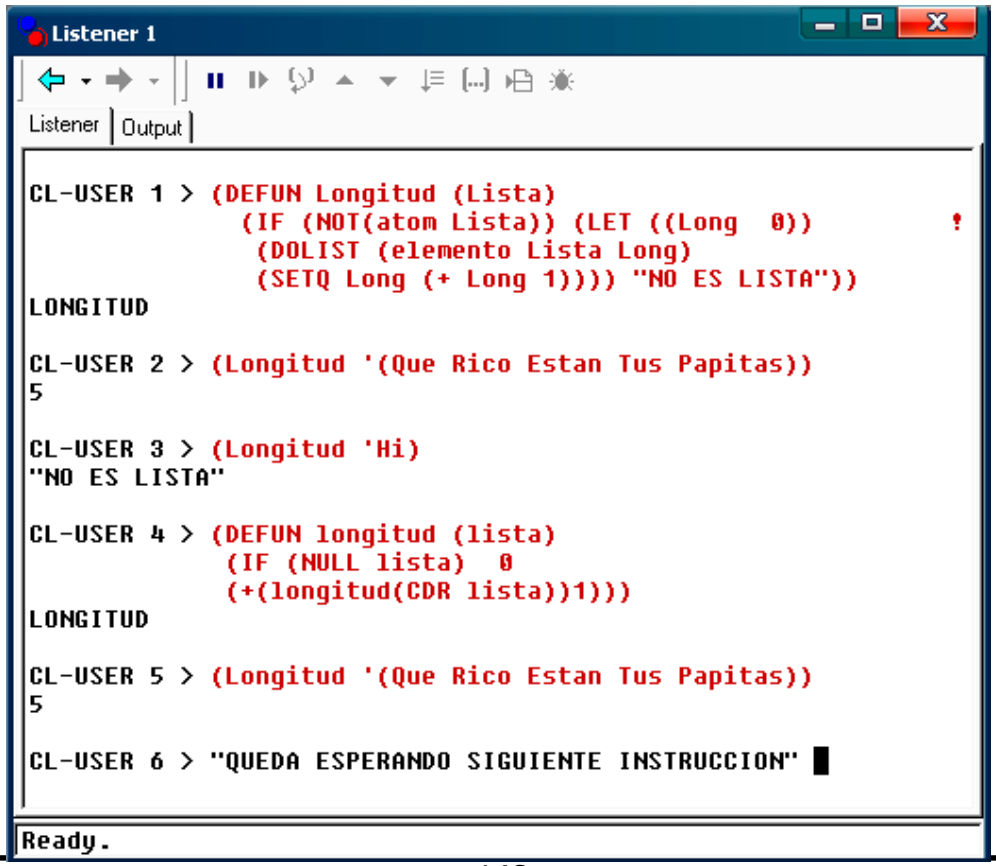
Ambos programas reciben como parámetro una lista cualquiera. Supongamos que tenemos la lista (A B C D E). Queremos probar el primer programa “primer-ultimo”. Se hace un FIRST de la lista quedando A como único elemento. En la siguiente línea se hace un REST de la lista y queda (B C D E) y se llama a la función “ultimo”. Ahora en la función “ultimo” se pregunta si el REST de la lista es “vacío” o no. Como no lo es se procede al siguiente paso el cual es un REST de la lista quedando ahora (C D E) y se llama de nuevo a “ultimo” recursivamente. Se vuelve a preguntar y no es vacío aun y haciendo otro REST queda (D E) y se llama de nuevo a “ultimo”. Se pregunta de nuevo y no es vacío y se prosigue al siguiente REST quedando (E). Se llama de nuevo a

"ultimo" y en el REST del IF la lista quedaría vacía entonces se devuelve el último parámetro el cual es la lista (E). Habiendo terminado allí se hace el CONS de "primer-ultimo" retornando la lista (A E) como respuesta.

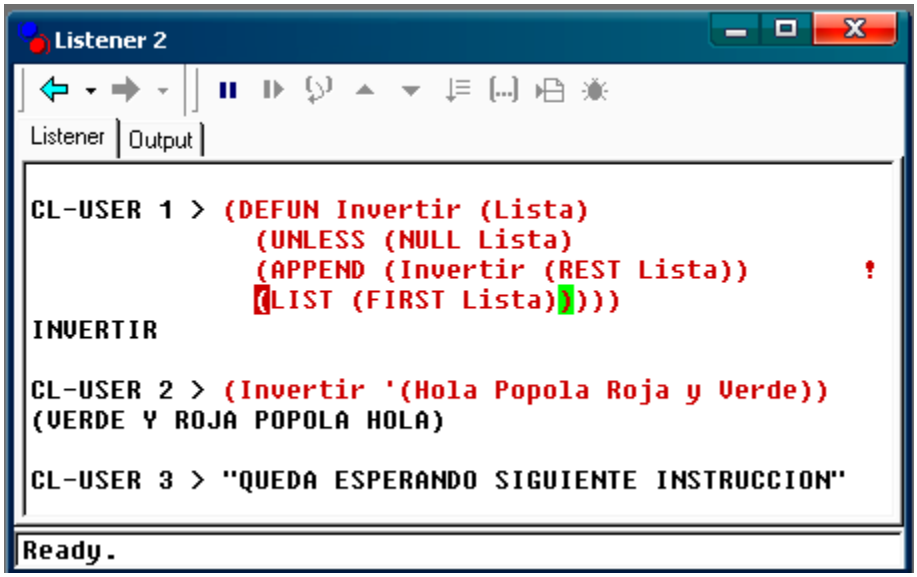
Solución 6-2



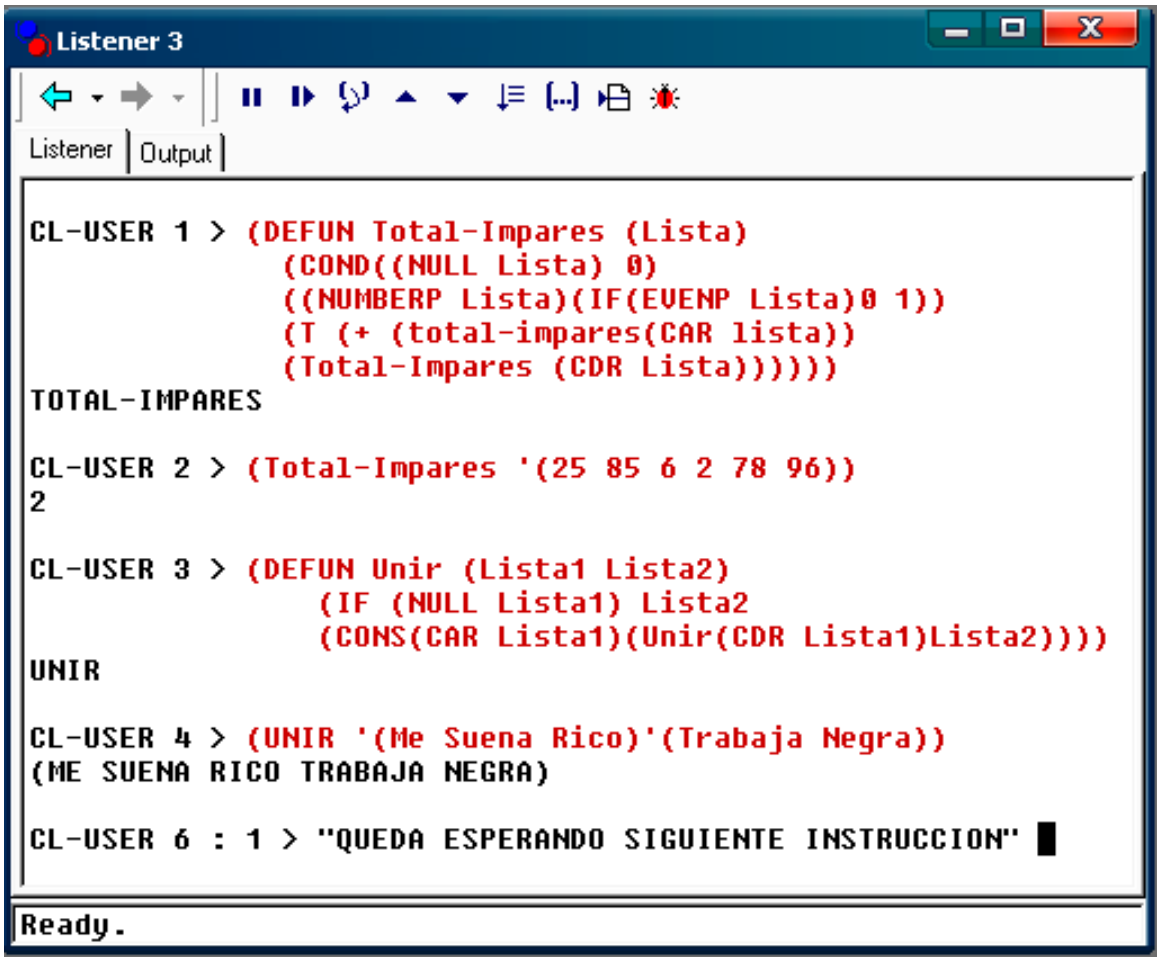
Solución 6-3



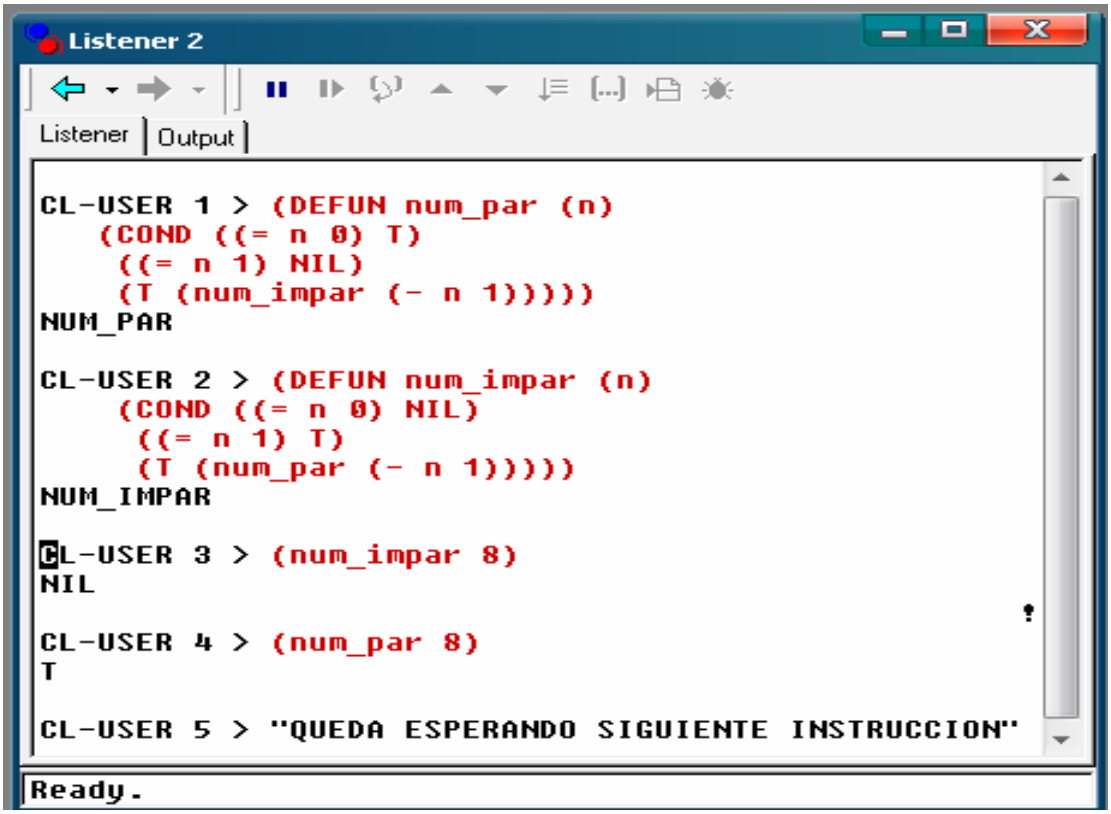
Solución 6-4



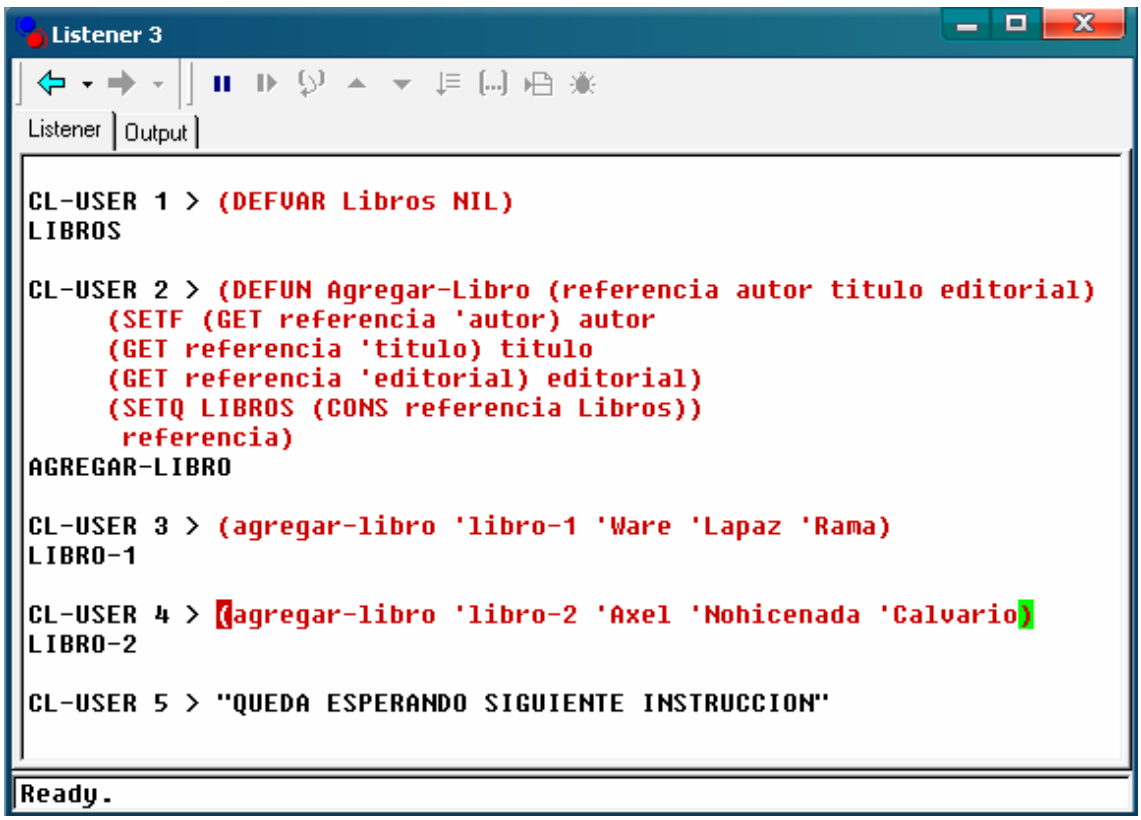
Solución 6-5, 6-6



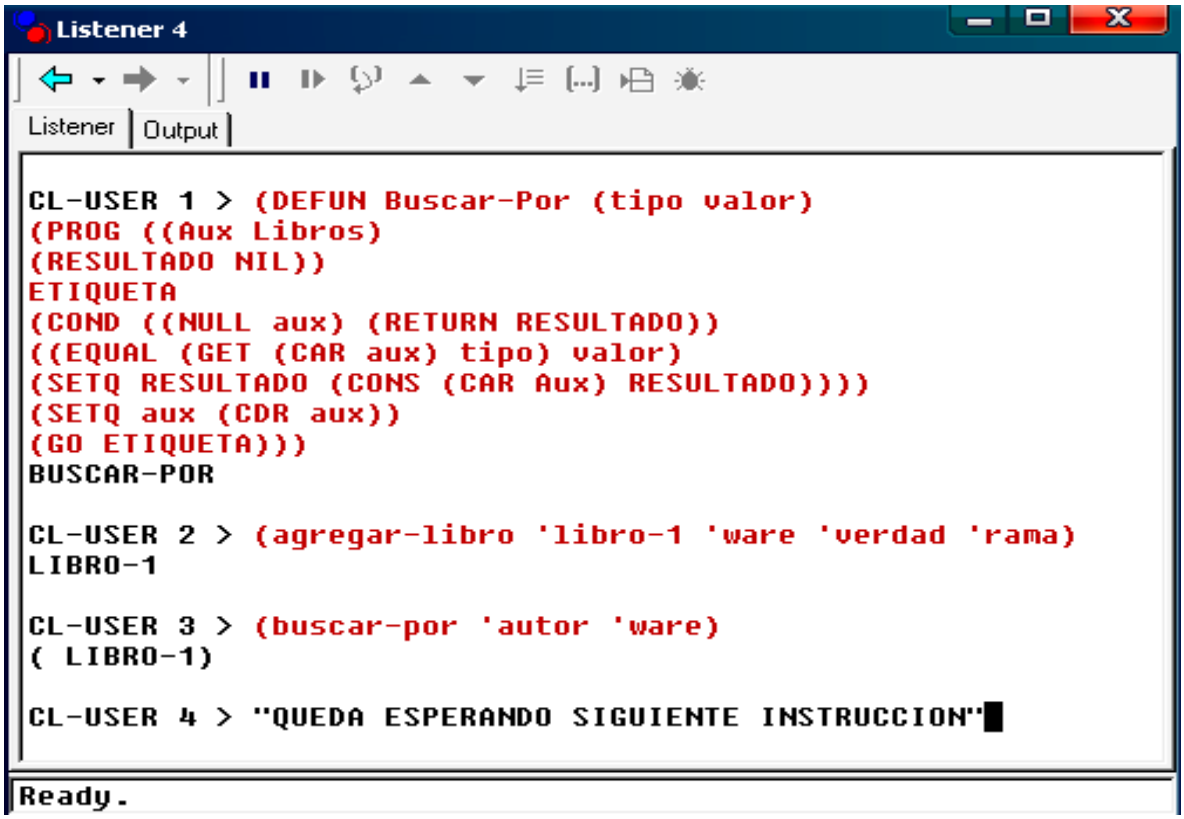
Solución 6-7



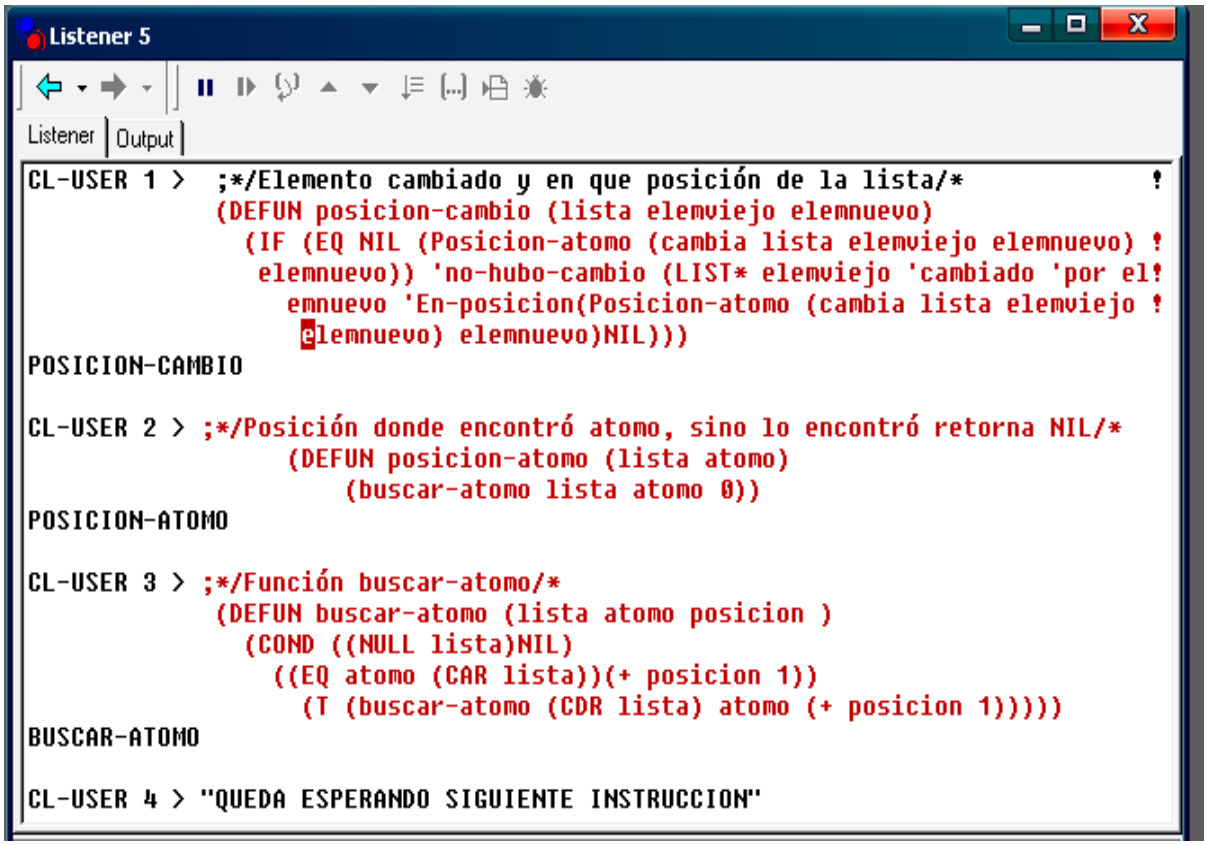
Solución 6-8

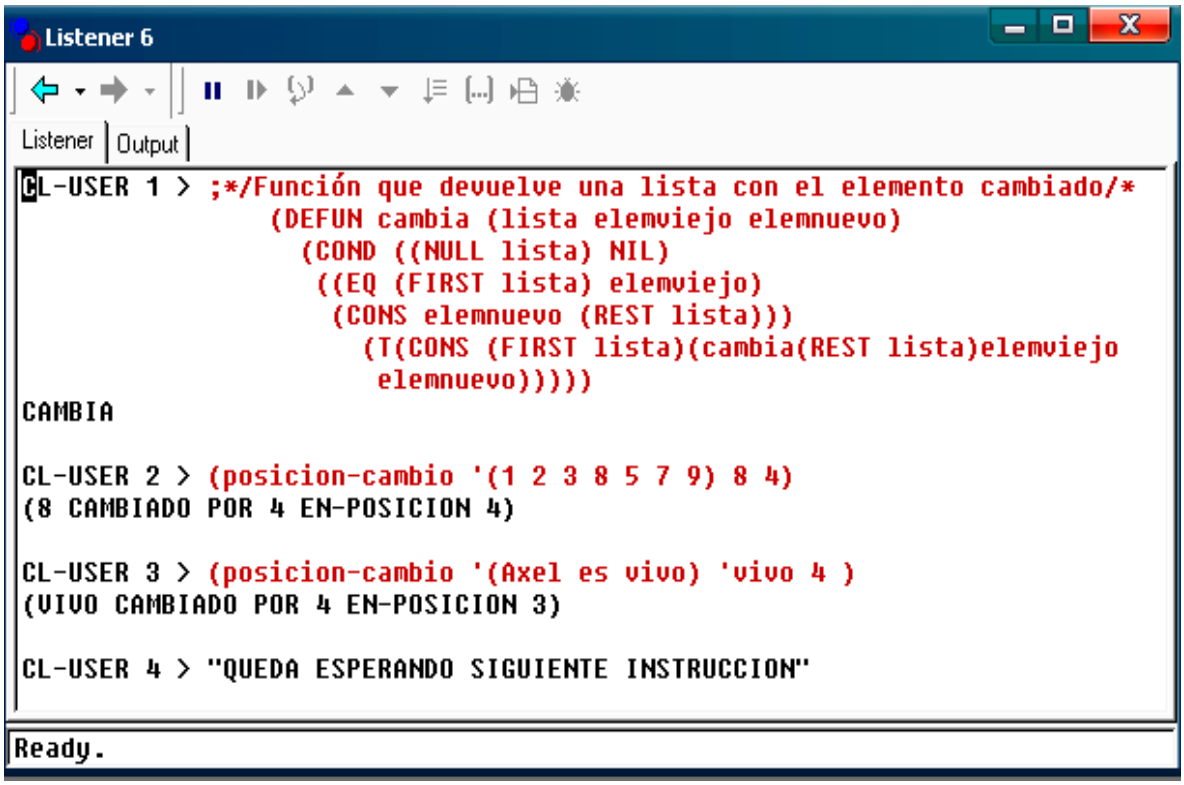


Solución 6-9



Solución 6-10




Listener 6
Listener
Output
CL-USER 1 > ;*/Función que devuelve una lista con el elemento cambiado/*
(DEFUN cambia (lista elemviejo elemnuevo)
(COND ((NULL lista) NIL)
((EQ (FIRST lista) elemviejo)
(CONS elemnuevo (REST lista)))
(T(CONS (FIRST lista)(cambia(REST lista)elemviejo
elemnuevo)))))
CAMBIA
CL-USER 2 > (posicion-cambio '(1 2 3 8 5 7 9) 8 4)
(8 CAMBIADO POR 4 EN-POSICION 4)
CL-USER 3 > (posicion-cambio '(Axel es vivo) 'vivo 4)
(VIVO CAMBIADO POR 4 EN-POSICION 3)
CL-USER 4 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
Ready.

SOLUCIÓN PRÁCTICA 7

Solución 7-1

a)

Se ejecuta el producto dando como resultado 15, luego el Print1 sabiendo que no hay salto de línea. Después el Print en línea aparte. Luego el Prin1 y como no hay salto de línea se coloca al lado del 2do. 15 con el espacio en blanco dado por el Print anterior. Por último el Print con salto de línea y su correspondiente valor devuelto por la llamada. En otras palabras lo que se muestra será:

```
15
15 15
15
15
```

b)

Primero muestra el mensaje, luego lee los caracteres introducidos, pero al llegar al CADR da error porque no se trata de una lista sino de un mensaje y un dato introducido. La solución es convertir esos datos en listas para luego ser extraídos con CADR. La solución seria:(print (list ‘tu ‘nombre ‘es (cadr *(list* (print ‘(¿como te llamas?))(read)))). Lo que muestre será:

```
(¿Cómo te llamas?) pepito
(Tu nombre es pepito)
(Tu nombre es pepito)
```

c)

Se evalúan los cuatro primeros Print incluyendo los saltos de línea. Se evalúa la suma de los números resultando 6 pero no se imprime hasta esperar que el Format sea evaluado y su correspondiente ~S que significa el valor 6. Evaluado el Format y sabiendo que NIL es su valor de retorno (debido a T), se procede a imprimir lo siguiente:

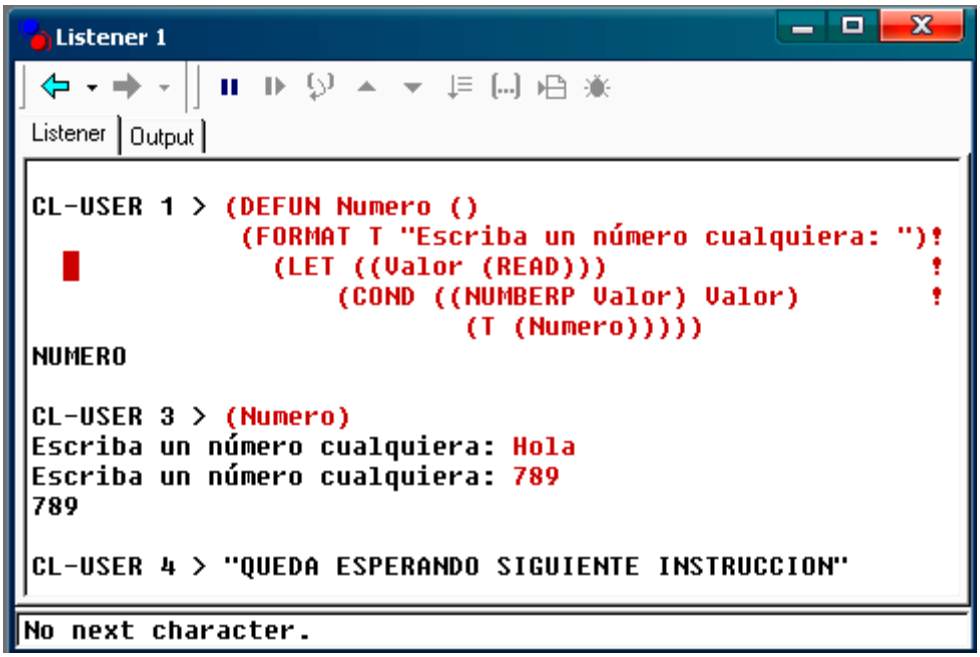
```
2
2
2
2 ¡Que onda! 6
NIL
```

d)

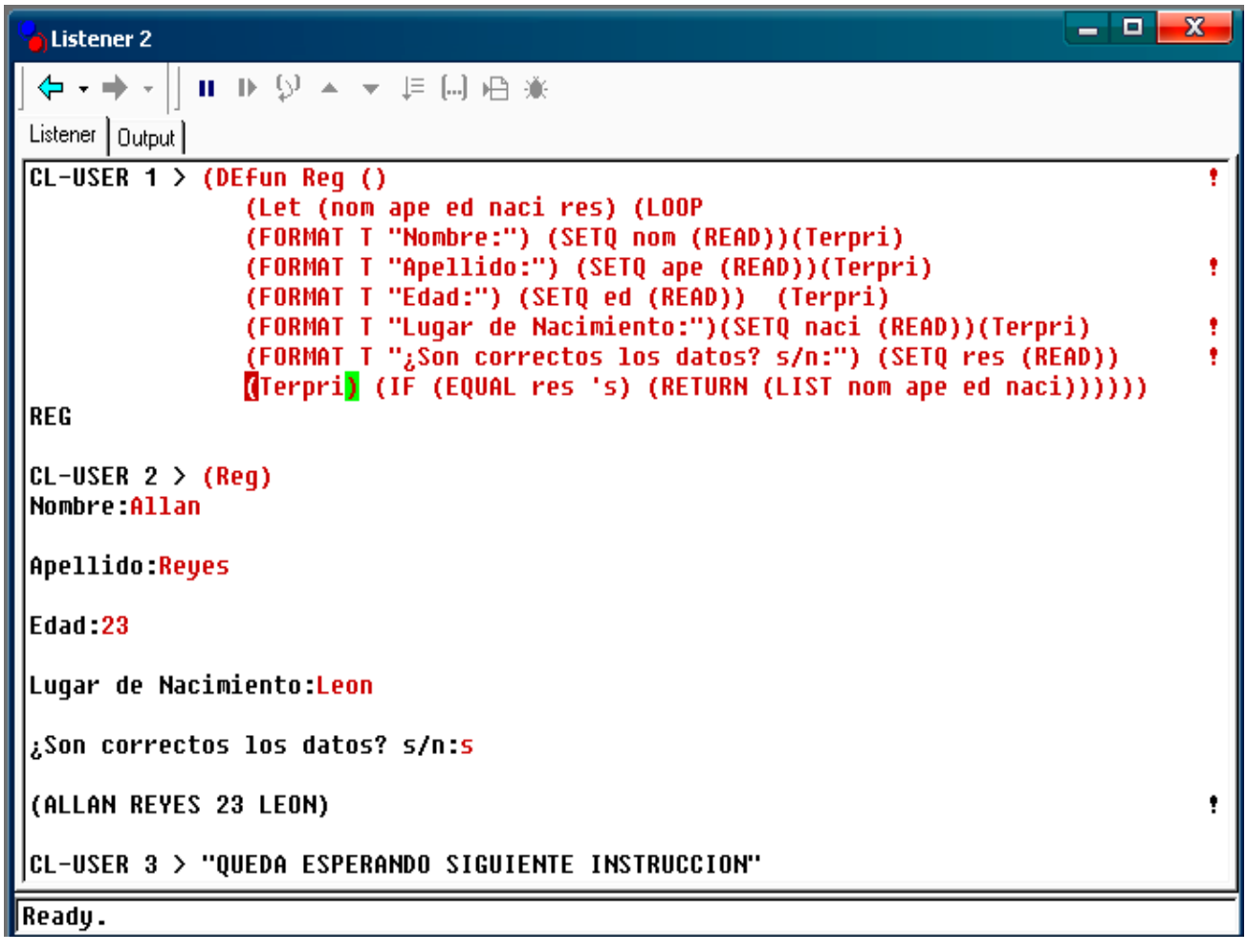
Se evalúan los cuatro primeros Print incluyendo los saltos de línea. Se evalúa la suma de los números resultando 17 pero no se imprime hasta esperar que el Format sea evaluado y su correspondiente ~S que significa el valor 17. Evaluado el Format y sabiendo que NIL es su valor de entrada (no retornará NIL) y sin omitir las comillas dobles por su formato, se procede a imprimir lo siguiente:

```
9
9
9
9
“!Como estas¡ 17”
^
```

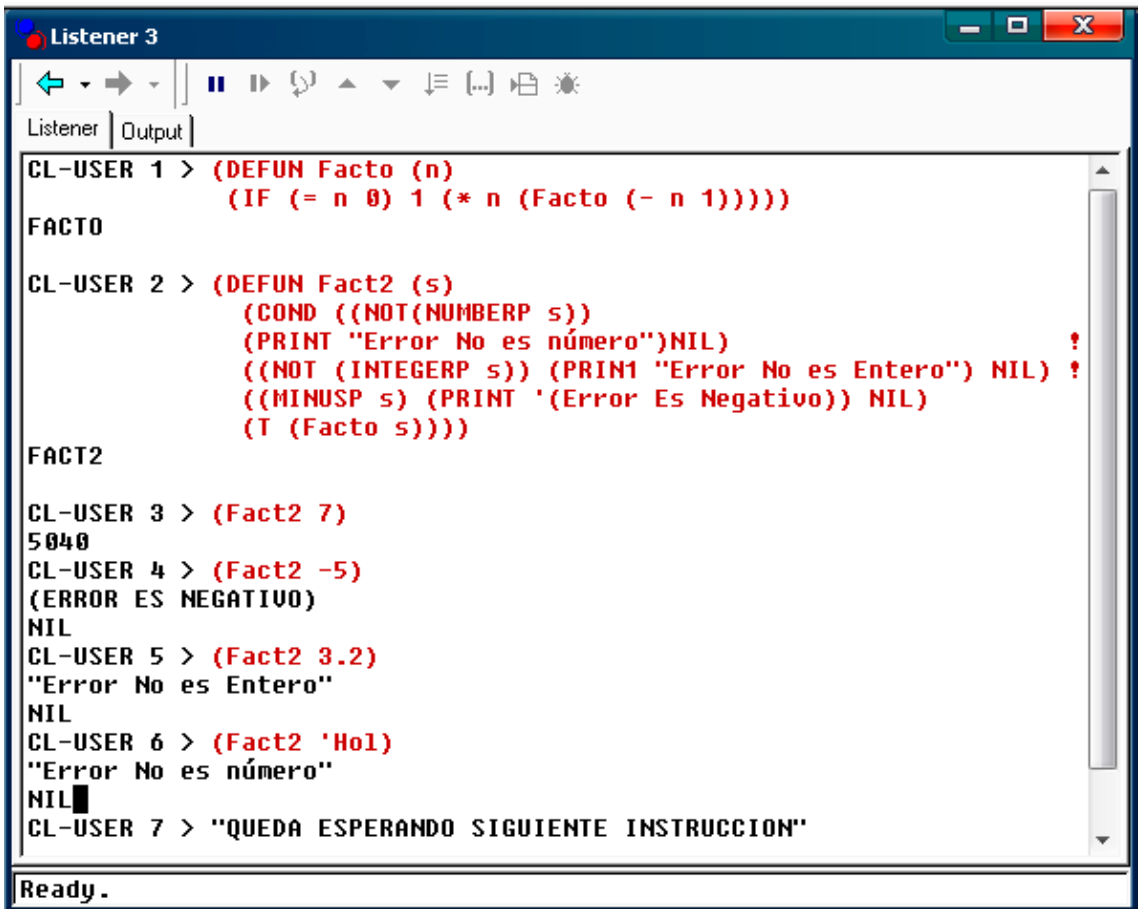
Solución 7-2



Solución 7-3



Solución 7-4



Solución 7-5

a.

Se ha creado una corriente para lectura de caracteres asociada al fichero ARTICULOS.DAT y se ha abierto. El primer READ ha leído el primer dato del archivo correspondiente al nombre del artículo y el segundo READ ha leído el precio de ese artículo.

b.

Se ha creado una corriente para escritura de caracteres asociada al fichero CLASES.DAT y se ha abierto. El format ha escrito en el archivo NOMBRE y NOTA dejando un salto de línea por lo tanto el contenido es ahora:

NOMBRE
NOTA
Y finalmente el fichero se ha cerrado.

c.

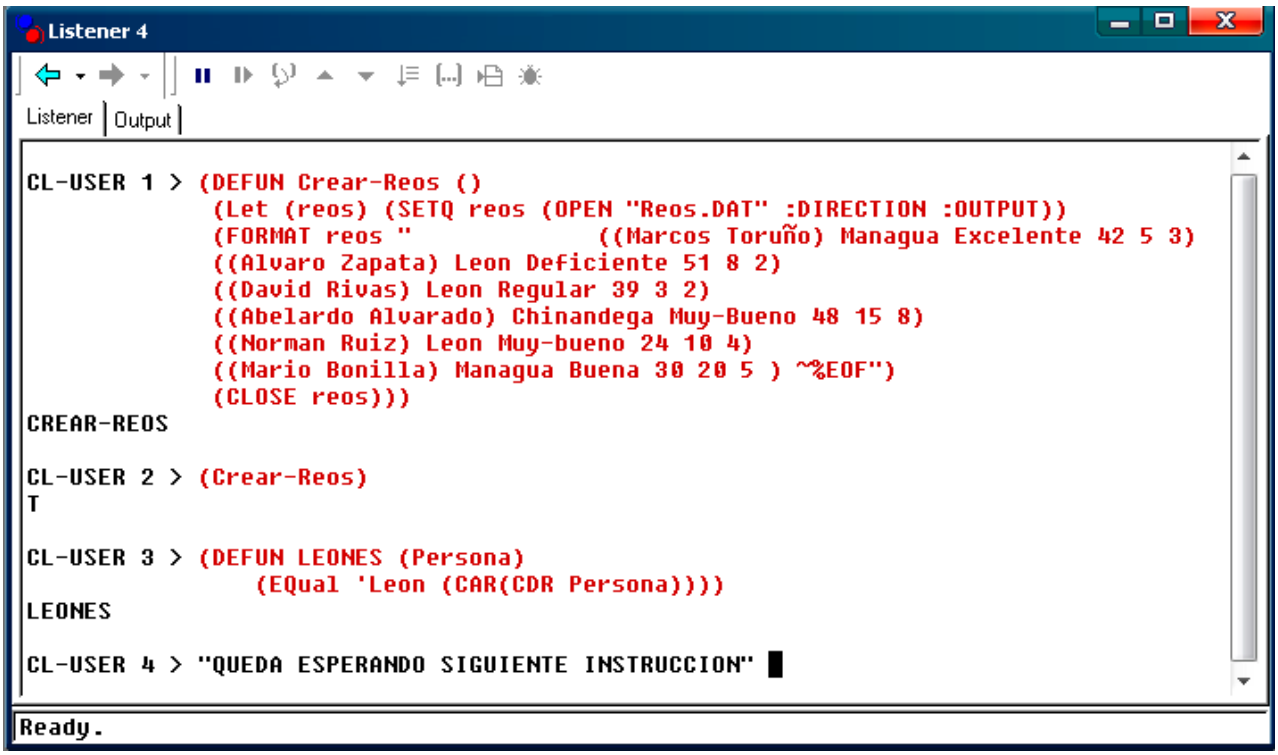
Se ha creado una corriente para escritura de caracteres asociada al fichero CLASES.DAT y se ha abierto. Se han ejecutado los READ hasta llegar al fin de fichero donde se ha producido un error por intentar leer más allá del fin de fichero.
El error se puede evitar, añadiendo más argumentos a READ.

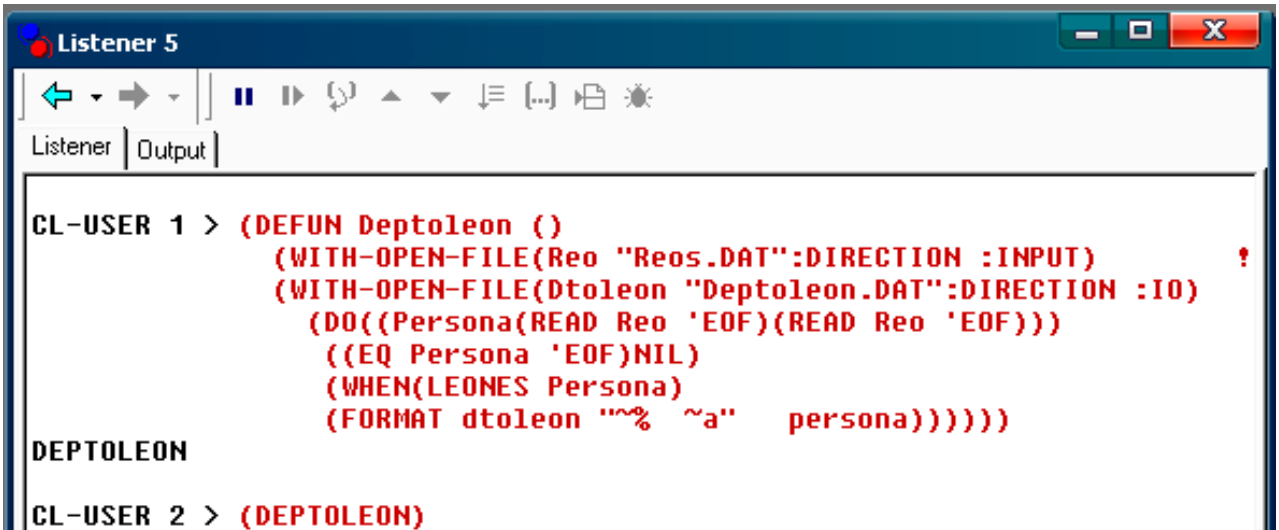
(READ expresión-c error-eof-p [valor-eof]) si error-eof-p es falso, no se produce el error de fin de fichero, sino que READ devuelve valor-eof (por defecto NIL) cuando intenta leer mas allá de fin de fichero.

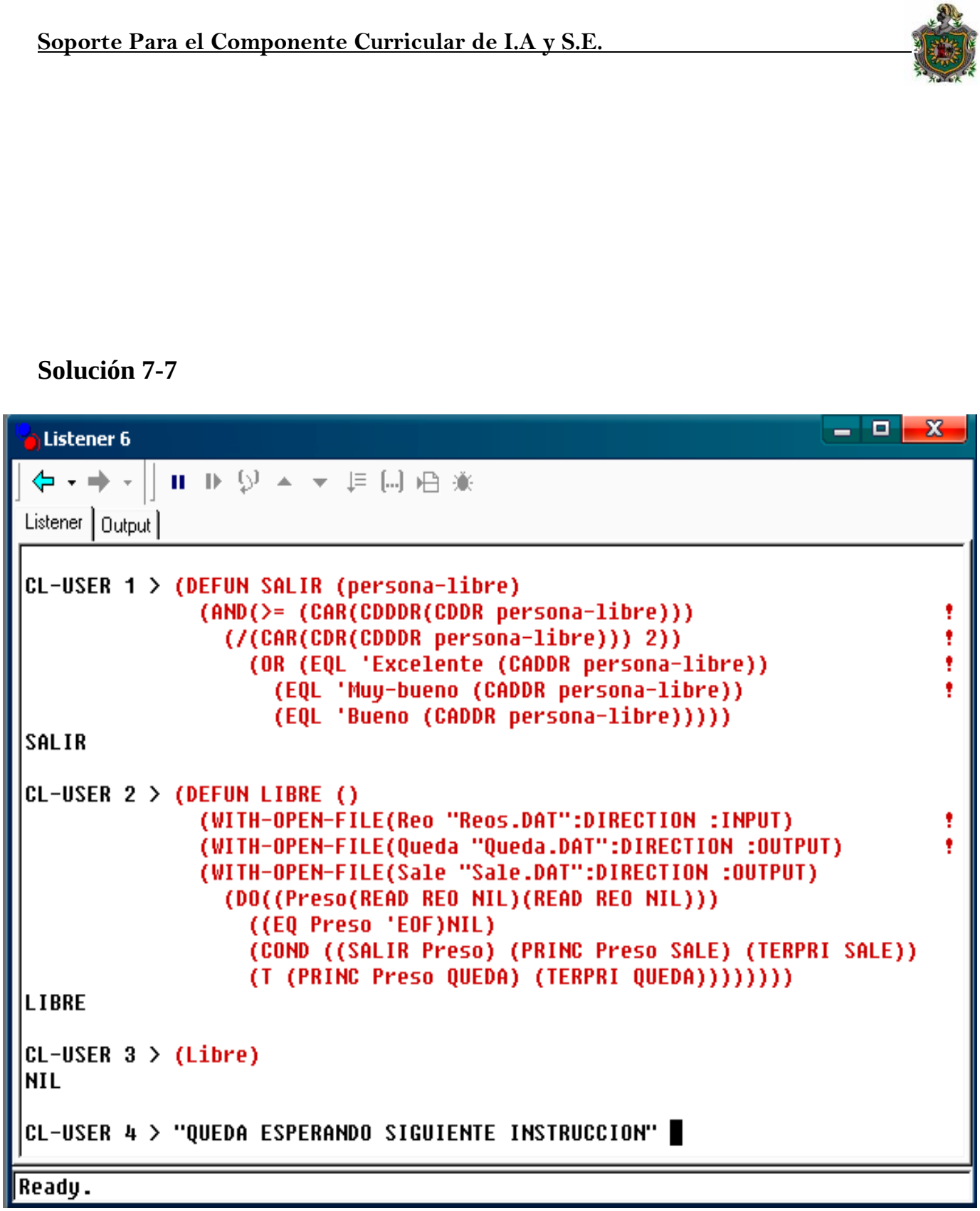
d.

Bucle infinito. No se produce error y READ se evalúa a Nil una vez alcanzado el fin de fichero.

Solución 7-6 Para comprobar mas específicamente de la creación de dichos archivos se abre la ruta C:\Documents and Settings\Snatch donde Snatch es el actual usuario de la Pc.







Solución 7-7

Listener 6

Listener | Output

```
CL-USER 1 > (DEFUN SALIR (persona-libre)
              (AND(>= (CAR(CDDDR(CDDR persona-libre)))
                (/(CAR(CDR(CDDDR persona-libre))) 2))
                  (OR (EQL 'Excelente (CADDR persona-libre))
                    (EQL 'Muy-bueno (CADDR persona-libre))
                    (EQL 'Bueno (CADDR persona-libre)))))
SALIR

CL-USER 2 > (DEFUN LIBRE ()
              (WITH-OPEN-FILE(Reo "Reos.DAT":DIRECTION :INPUT)
              (WITH-OPEN-FILE(Queda "Queda.DAT":DIRECTION :OUTPUT)
              (WITH-OPEN-FILE(Sale "Sale.DAT":DIRECTION :OUTPUT)
                (DO((Preso(READ REO NIL)(READ REO NIL)))
                  ((EQ Preso 'EOF)NIL)
                  (COND ((SALIR Preso) (PRINC Preso SALE) (TERPRI SALE))
                  (T (PRINC Preso QUEDA) (TERPRI QUEDA)))))))))
LIBRE

CL-USER 3 > (Libre)
NIL

CL-USER 4 > "QUEDA ESPERANDO SIGUIENTE INSTRUCCION"
```

Ready.

GLOSARIO

Abstracción de Procedimientos: Proceso de ocultar detalles mediante niveles de procedimiento y subprocedimientos.

Barrera de Abstracción: Cerca virtual producida por atracción de procedimientos u atracción de datos tras la cual se ocultan los detalles de nivel inferior.

Celda Cons: Entidad bidireccional a partir de la cual se construyen las listas en memoria del computador.

Cerca Virtual: Cerca que se levanta a llamar a los procedimientos o primitivas que ligan variables.

Comentario: Anotación de un procedimiento como ayuda para programadores ignorada por los intérpretes y los compiladores.

Efecto Secundario: Todo aquello que hace un procedimiento y que persiste después de que devuelve su valor.

Elemento de nivel superior: El elemento de lista que está directamente contenido en esa lista. Debe distinguirse de los elementos insertados en sublistas.

Explosión combinatoria: Incremento acelerado en el número de maneras posibles de elegir combinaciones alternativas de elementos conforme el número de estos crecen.

Flujo de entrada: Un objeto Lisp que proporciona datos, casi siempre conectado a un archivo o el teclado.

Flujo de salida: Un objeto Lisp que consume datos, con frecuencia conectado a un archivo o la pantalla.

Intérprete: Programa que realiza la acción especificada por definiciones expresadas en forma de texto, ejecutando forma por forma las acciones que son llamadas en el cuerpo de esas definiciones.

Pila: Registro dinámico de los procedimientos y primitivas que ligan variables que se han llamado pero que aún no sea devuelto un valor.

Primitiva: Procedimiento construido internamente.

Procedimiento: Especificación pasó paso de cómo hacer algo expresada en un lenguaje de programación, como Lisp.

Recolección de Basura: Proceso de recuperar la memoria para uso posterior una vez que ésta ha dejado estar accesible mediante cualquier procedimiento o variable.

BIBLIOGRAFÍA

1. ALLEN, J. Anatomy of LISP. MacGraw-Hill, New York.

2. LISP: The Language of Artificial Intelligence. Collins, London.

3. Lisp: Tercera Edision, Patrick Henry, Berthold Klaus.

5. Introducción al LISP (El lenguaje básico para la Inteligencia Artificial) FARRENY, H. Masson, Barcelona.

6. C. Programación en LISP Paraninfo, QUEINNEC, Madrid.

7. Internet
http://es.wikipedia.org/wiki/Sistema_experto

(http://www.ia.uned.es/~jgb/)

http://.go.to/inteligenciartificial

www.itba.edu.ar

www.publispain.com.supertutorial