Universidad Nacional Autónoma de Nicaragua
Unan - León
Facultad de Ciencias y Tecnología
Departamento de Computación

Tesis para optar al título de Ingeniero en Sistemas de Información

Tema:

SOPORTE PARA EL COMPONENTE CURRICULAR DE INTELIGENCIA ARTIFICIAL Y SISTEMAS EXPERTOS BASADO EN EL LENGUAJE DE PROGRAMACION PROLOG CON EL EDITOR SWI-PROLOG-EDITOR 5.6.48.

Integrantes:

- **Br. Gerald Antonio Gutiérrez Soriano.**
- **Br. Wilfredo Ramiro Herrera Serrano.**
- **Br. Darwin Francisco Soto González.**

Tutor: Ing. Karina Esquivel.

León, Jueves 15 de enero de 2009.

INDICE DE CONTENIDO

AGRADECIMIENTO

- A Dios nuestro Señor que nos ha guiado y nos ha dado sabiduría para alcanzar esta meta.

- A nuestros padres y familiares que con su apoyo y comprensión hicieron posible este logro en nuestras vidas.

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

I. INTRODUCCIÓN

La Inteligencia Artificial surgió como el resultado de la investigación en lógica matemática, se ha enfocado en la simulación del trabajo mental y la construcción de algoritmos para solución a problemas de propósito general, un campo de ésta son los Sistemas Expertos que juega un papel muy importante en el desarrollo de sistemas tecnológicos y avanzados que facilitan al experto humano un sin número de tareas en diferentes facetas de la vida.

Actualmente existen muchos lenguajes de programación utilizados en Inteligencia Artificial, como Prolog, Linksys, Lisp. El presente trabajo monográfico es un soporte basado en el lenguaje Prolog, proveniente del francés Programation et Logique que fue desarrollado por la Universidad de Marsella como una herramienta práctica para programación lógica.

El soporte proporciona una serie de temas referentes a los conceptos más importantes de la Inteligencia Artificial con el que se pretende que el alumno se familiarice, tomando en cuenta, que este debe tener ciertos conocimientos previos en programación para un mejor entendimiento del mismo.

II. ANTECEDENTES

El estudio de la Inteligencia Artificial es una de las disciplinas más antiguas, por más de 2000 años los filósofos no han escatimado esfuerzo alguno por comprender como se ve, recuerda y razona, junto con la forma en que estas actividades deberían realizarse. La Inteligencia Artificial nació en 1943 cuando Warren McCulloch y Walter Pitts propusieron un modelo de neurona del cerebro humano y animal, pero fue a mediados de 1950 en que comenzaron los primeros desarrollos en Inteligencia Artificial con el trabajo de Alan Turing, a partir de lo cual la ciencia ha pasado por una serie de cambios.

El término de Inteligencia Artificial fue fortalecido formalmente en 1956 por Jon McCarthy, Marvin Minsky y Claude Shannon en la Conferencia de Darthmounth, un congreso en el que se hicieron previsiones triunfalistas a diez años que jamás se cumplieron, lo que provocó el abandono casi total de las investigaciones durante quince años. En los años 70, un equipo de investigadores dirigido por Edward Feigenbaum comenzó a elaborar un proyecto para resolver problemas de la vida cotidiana, es así como nacen los Sistemas Expertos, el primer sistema experto fue el denominado Dendral, un intérprete de espectrograma de masa construido en 1967, pero el más influyente resultaría ser el Mycin de 1974. Ya en los años 80, se desarrollaron lenguajes especiales para utilizar con la Inteligencia Artificial, tales como el LISP o PROLOG.

Debido a la aplicación de esta disciplina en diversas áreas de la computación hemos desarrollado este soporte en conjunto con el Departamento de Computación de la UNAN- LEON, el cual oferta a los estudiantes de la Carrera de Ingeniería en Sistemas de Información, la asignatura de Inteligencia Artificial y Sistemas Expertos en el IX semestre del plan de estudio actual (I semestre de V año de la carrera) que actualmente cuenta con un soporte basado en Lisp utilizando el editor Xanalys Lispworks 4.3 que sirve de guía para los estudiantes que deseen inscribir el componente curricular.

III. JUSTIFICACIÓN

La Inteligencia Artificial es una de las áreas de las ciencias computacionales encargadas de la creación de hardware y software que tengan comportamientos inteligentes. La Inteligencia Artificial trata de enfocar el concepto de Inteligencia en las máquinas.

Con el presente trabajo pretendemos dar a conocer las ventajas y desventajas que tiene el lenguaje de programación Prolog en la Inteligencia Artificial con respecto al lenguaje de programación Lisp, es de vital importancia mencionar que nos motivamos a realizarlo para que sirva de ayuda tanto a los profesores como los alumnos y así puedan obtener los conocimientos básicos sobre Inteligencia Artificial y Sistemas Expertos, los cuales serán afianzados con prácticas que se relacionan con lo expuesto en la teoría y de esa manera a través de la práctica aprendan a utilizar el potencial del Lenguaje de programación Prolog.

IV. OBJETIVOS

Objetivo General:

- Elaborar un soporte para el componente curricular de Inteligencia Artificial y Sistemas Expertos basado en el lenguaje de programación Prolog que sirva de guía para impartir dicha asignatura.

Objetivos Específicos:

- Explicar los conceptos y características principales sobre la cuales se fundamenta la Inteligencia Artificial y los Sistemas Expertos.

- Permitir una mejor planificación y organización de la asignatura.

- Dar a conocer las ventajas y desventajas que tiene el lenguaje de programación Prolog en la Inteligencia Artificial con respecto al lenguaje de programación Lisp.

V. SITUACIÓN ACTUAL DE LA ASIGNATURA

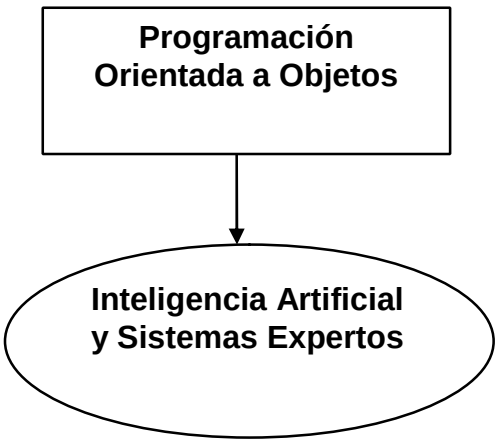
La asignatura de Inteligencia Artificial y Sistemas Expertos, se imparte en la carrera de Ingeniería en Sistemas de Información de la UNAN–LEÓN, que ofrece el Departamento de Computación de la Facultad de Ciencias y Tecnología de dicha Universidad.

La carrera se desarrolla a lo largo de 10 semestres de estudios, es decir 5 años académicos para la culminación de la misma. Esta asignatura se imparte en el IX semestre del plan de estudios actual y consta de 3 créditos. El período lectivo para esta asignatura consta de 4 horas a la semana, de las cuales 2 horas se dedican a la parte teórica y 2 horas a la parte práctica. El total de semanas de que consta un semestre es de 16, logrando así un total de 64 horas, de las cuales 32 horas son teóricas y 32 horas son prácticas (laboratorio).

	Horas semanales	**Horas semestrales**
Teoría	2 horas	32 horas
Practica	2 horas	32 horas
Total	**4 horas**	**64 horas**

VI. RELACIÓN CON OTRAS ASIGNATURAS

La asignatura de Inteligencia Artificial y Sistemas Expertos tiene como prerrequisito la asignatura de Programación Orientada a Objetos la cual se imparte en el VII Semestre.



Programación Orientada a Objetos

El objetivo fundamental de esta asignatura es introducir al estudiante en los conceptos básicos de programación orientada a objetos ya que la mayoría de los actuales lenguajes de programación se basan en esta metodología empleando abstracción, encapsulamiento, herencia y polimorfismo, necesarios para asignaturas como Programación Visual I y II, Ingeniería del Software.

El contenido por temas de la asignatura es el siguiente:

Tema 1: Introducción a la Programación Orientada a Objetos (POO)
En este tema se estudian los conceptos generales en los cuales se basa la metodología de programación orientada a objetos como: objeto, método, clase, subclase, etc.

Tema 2: Lenguaje C++
En este segundo tema se abordan las características del lenguaje C++ y una comparación entre lenguaje C y el lenguaje C++ estableciendo similitudes y diferencias.

Tema 3: Clases
En este tema se describen los conceptos de clase, miembros de una clase, control de acceso a los miembros de una clase, implementación de una clase, el puntero this, funciones miembros y objetos constantes, constructor, constructor copia, destructor.

Tema 4: Trabajando con clases
En este tema se abordan conceptos más avanzados de clases para utilizarlos en la solución de problemas: objetos miembros de una clase, acceso a datos miembros

privados, clases en ficheros de cabecera, miembros static de una clase, punteros a miembros de una clase, funciones amigas de una clase y clases internas.

Tema 5: Operadores sobrecargados
En este se estudia el concepto de sobrecarga de un operador (binario, unario, asignación, indexación, operador función) y su utilidad en la resolución de problemas.

Tema 6: Clases Derivadas
En este sexto tema se abordan los conceptos de herencia entre clases y polimorfismo haciendo énfasis en la resolución de problemas a través de clases bases, clases derivadas, herencia simple, constructores y destructores de clases derivadas, funciones, constructores y destructores de clases virtuales, clases abstractas, herencia múltiple y virtual.

Tema 7: Tipos Genéricos
En este tema se abordan los conceptos de funciones y clases genéricas, argumentos de un patrón, instancias de plantillas con tipos definidos por el usuario, funciones que usan plantillas como parámetros y amigos de plantillas.

Tema 8: Manejo de Excepciones
En este tema se aborda el mecanismo que debe seguirse para manejar los errores que puedan ocurrir durante la ejecución de un programa mediante el lanzamiento de excepciones.

Todos los contenidos teóricos de esta asignatura son ejercitados en las prácticas de laboratorio.

Esta asignatura aporta los conocimientos de Programación Orientada a Objetos, necesarios para comprender los lenguajes orientados a objetos, utilizados comúnmente en la Base de hechos de los lenguajes lógicos.

VII. CONTENIDO DE LA ASIGNATURA

Tema 1: Introducción a la Inteligencia Artificial y los Sistemas Expertos

1.1. Introducción a la Inteligencia Artificial.
1.1.1. Historia de la Inteligencia Artificial.
1.1.2. Definición de Inteligencia Artificial.
1.1.3. Características de la Inteligencia Artificial.
1.1.4. Prueba de Turing.
1.1.5. Principales ramas de la Inteligencia Artificial.
1.1.6. Lenguajes de programación.
1.1.6.1. Lenguajes de Inteligencia Artificial.
1.1.6.2. Ventajas y desventajas entre Prolog y LISP.
1.1.7. Sistemas de desarrollo.
1.1.8. Ciencias que aportan a la Inteligencia Artificial.
1.2. Introducción a los Sistemas Expertos.
1.2.1. Definición de los Sistemas Expertos.
1.2.2. Antecedentes de los Sistemas Expertos.
1.2.3. Estructura básica de un Sistema Experto.
1.2.4. Tipos de Sistemas Expertos.
1.2.5. Ventajas y limitaciones de los Sistemas Expertos.
1.2.6. Tareas que realiza un Sistema Experto.
1.2.7. Ejemplos de Sistemas Expertos.

Tema 2: Introducción al lenguaje de programación Prolog

2.1. Introducción a Prolog.
2.2. Historia.
2.3. Descripción.
2.4. Predicados en Prolog.
2.4.1. Variables.
2.4.1.1. Cómo se asignan valores a las variables.
2.4.1.2. Variables anónimas.
2.4.2. Reglas.
2.4.3. Términos en Prolog.
2.5. Estructura de un programa Prolog.
2.6. Preguntas.
2.6.1. El mecanismo de unificación.
2.7. Tipos de datos en Prolog.
2.7.1. Operadores
2.8. Estructura de datos
2.9. Recursividad.
2.9.1. Definición de recursividad.
2.10. Entrada y salida de datos.
2.10.1. Lectura y escritura de términos.
2.10.2. Escritura con formato.
2.10.3. Lectura y escritura de caracteres.

2.11. Desarrollo de versiones actuales.
2.12. Ejercicios resueltos.

Tema 3: Listas en Prolog

3.1. Definición de lista.
3.2. Estructura con lista.
3.2.1. Simbología
3.3. Operaciones sobre lista.
3.3.1. Construcción de lista.
3.3.2. Inserción de un elemento.
3.3.3. Primer elemento.
3.3.4. Ultimo elemento.
3.3.5. Penúltimo elemento.
3.3.6. Resto de una lista.
3.3.7. Relación de pertenencia.
3.3.8. Selección de un elemento.
3.3.9. Concatenación de lista.
3.3.10. Lista inversa.
3.3.11. Palíndromo.
3.3.12. Permutación.
3.3.13. Rotación de un elemento.
3.3.14. Sublista.
3.3.15. Subconjunto de una lista.
3.3.16. Lista con todos los elementos iguales.
3.3.17. Paridad de la longitud de una lista.
3.4. Aritmética con lista.
3.4.1. Suma.
3.4.2. Producto.
3.4.3. Longitud de lista.
3.5. Ejercicios resueltos.

Tema 4: Estructuras de control

4.1. Reevaluación.
4.1.1. Introducción.
4.1.2. Definición de reevaluación.
4.1.3. Característica de la reevaluación.
4.1.4. Conceptos relacionados a la reevaluación.
4.2. Predicado Corte (!).
4.2.1. Introducción.
4.2.2. Definición Corte.
4.2.3. Representación del Corte.
4.2.4. Efecto del predicado Corte.
4.2.5. Utilidad del Corte.
4.2.6. Ventajas y desventajas del Corte.
4.3. Predicado Fallo (fail).

4.3.1. Introducción.
4.3.2. Definición de Fail.
4.3.3. Utilidad de Fail.
4.3.4. Ventajas y desventajas del uso de Fail.
4.4. Predicado Not (Negación).
4.4.1. Introducción.
4.4.2. Definición.
4.4.3. Utilidad de Not.
4.4.4. Ventajas y desventajas del uso de Not.
4.5. Ejercicios resueltos.

Tema 5: Árboles

5.1. Definición de árboles.
5.2. Representación de árboles.
5.3. Características de árboles.
5.4. Recorrido de árboles.
5.5. Árboles binarios.
5.5.1. Árboles binarios de búsqueda
5.6. Ejercicios resueltos.

Tema 6: Programación Lógica y Base de Datos

6.1. Definición de Base de Datos.
6.2. Tipos de Base de Datos.
6.2.1. Base de Datos Lógicas.
6.2.2. Base de Datos Orientadas a Objetos.
6.2.3. Base de Datos Relacionales.
6.3. Álgebra Relacional.
6.4. Representación de las relaciones en Prolog.
6.5. Operaciones fundamentales.
6.5.1. Selección.
6.5.2. Proyección.
6.5.3. Unión.
6.5.4. Diferencia de conjuntos.
6.5.5. Producto cartesiano.
6.5.6. Intersección de conjuntos.
6.5.7. La operación reunión natural.
6.5.8. División.
6.6. Ejercicios resueltos.

VIII. DESARROLLO DEL TEMARIO

Tema 1: Introducción a la Inteligencia Artificial y los Sistemas Expertos.

En esta primera unidad se pretende que el alumno conozca un poco de historia y se familiarice con los conceptos que se encuentran ligados a la Inteligencia Artificial y los Sistemas Expertos.

La Inteligencia Artificial es una combinación de la ciencia del computador, fisiología y filosofía, tan general y amplio como eso, es que reúne varios campos (robótica, sistemas expertos, por ejemplo), todos los cuales tienen en común la creación de máquinas que simulan el conocimiento inteligente del experto humano.

Contenido:

1. Introducción a la Inteligencia Artificial y los Sistemas Expertos.
1.1. Introducción a la Inteligencia Artificial.
1.1.1. Historia de la Inteligencia Artificial.
1.1.2. Definición de Inteligencia Artificial.
1.1.3. Características de la Inteligencia Artificial.
1.1.4. Prueba de Turing.
1.1.5. Principales ramas de la Inteligencia Artificial.
1.1.6. Lenguajes de programación.
1.1.6.1. Lenguajes de Inteligencia Artificial.
1.1.6.2. Ventajas y desventajas entre Prolog y LISP.
1.1.7. Sistemas de desarrollo.
1.1.8. Ciencias que aportan a la Inteligencia Artificial.
1.2. Introducción a los Sistemas Expertos.
1.2.1. Definición de los Sistemas Expertos.
1.2.2. Antecedentes de los Sistemas Expertos.
1.2.3. Estructura básica de un Sistema Experto.
1.2.4. Tipos de Sistemas Expertos.
1.2.5. Ventajas y limitaciones de los Sistemas Expertos.
1.2.6. Tareas que realiza un Sistema Experto.
1.2.7. Ejemplos de Sistemas Expertos.

1. Introducción a la Inteligencia Artificial y los Sistemas Expertos

1.1. Introducción a la Inteligencia Artificial

1.1.1. Historia de la Inteligencia Artificial

Desde sus comienzos hasta la actualidad, la Inteligencia Artificial ha tenido que hacer frente a una serie de problemas:

- Los computadores no contienen verdaderos significados.
- Los computadores no tienen autoconciencia (emociones, sociabilidad, etc.).
- Un computador sólo puede hacer aquello para lo que está programado.
- Las máquinas no pueden pensar realmente.

En 1843, Lady Ada Augusta Byron, patrocinadora de Charles Babbage planteó el asunto de si la máquina de Babbage podía "pensar". Los primeros problemas que se trató de resolver fueron puzzles, juegos de ajedrez, traducción de textos a otro idioma. Durante la II Guerra Mundial Norbert Wiener y John Von Neumann establecieron los principios de la cibernética en relación con la realización de decisiones complejas y control de funciones en máquinas.

En 1937 el matemático inglés Alan Mathison Turing (1912-1953) publicó un artículo de bastante repercusión sobre los “Números Calculables”, que pueden considerarse el origen oficial de la Informática Teórica.

En este artículo, introdujo la Máquina de Turing, una entidad matemática abstracta que formalizó el concepto de algoritmo y resultó ser la precursora de las computadoras digitales. Con ayuda de su máquina pudo demostrar que existen problemas difíciles que ningún ordenador será capaz de solucionar, por ello Turing es considerado el padre de la teoría de la computación.

También se le considera el padre de la Inteligencia Artificial, por su famosa Prueba de Turing, que permitiría comprobar si un programa de ordenador puede ser tan inteligente como un ser humano.

La teoría de la retroalimentación en mecanismos, como por ejemplo un termostato que regula la temperatura en una casa, tuvo mucha influencia. Esto aún no era propiamente Inteligencia Artificial. Se hizo mucho en traducciones (Andrew Booth y Warren Weaver), lo que sembró la semilla hacia el entendimiento del lenguaje natural. En el año 1955 Herbert Simón, el físico Allen Newell y J.C. Shaw, programador de la RAND Corp. y compañero de Newell, desarrolla el primer lenguaje de programación orientado a la resolución de problemas de la Inteligencia Artificial, el IPL-11. Un año más tarde estos tres científicos desarrollan el primer programa de Inteligencia Artificial al que llamaron Logic Theorist, el cual era capaz de demostrar teoremas matemáticos, representando cada problema como un modelo de árbol, en el que se seguían ramas en busca de la solución correcta, que

resultó crucial. Este programa demostró 38 de los 52 teoremas del segundo capítulo de Principia Mathematica de Russel y Whitehead.

En 1956, con la ahora famosa conferencia de Dartmouth, organizada por John McCarthy y en la que se utilizó el nombre de Inteligencia Artificial para este nuevo campo, el cual se separó de la ciencia del computador, como tal. Se estableció como conclusión fundamental la posibilidad de simular inteligencia humana en una máquina.

En 1957 Newell y Simón continúan su trabajo con el desarrollo del GPS, el cual era un sistema orientado a la resolución de problemas; a diferencia del Logic Theorist, el cual se orientó a la demostración de teoremas matemáticos, GPS no estaba programado para resolver problemas de un determinado tipo, razón a la cual debe su nombre. Resuelve una gran cantidad de problemas de sentido común, como una extensión del principio de retroalimentación de Wiener (Procesos mediante los cuales un sistema abierto recoge información sobre los efectos de sus decisiones internas en el medio, información que actúa sobre las decisiones sucesivas).

Diversos centros de investigación se establecieron, entre los más relevantes están, la Universidad Carnegie Mellon, el Massachusetts Institute of Technologie (MIT), encabezado por Marvin Minsky, la Universidad de Standford e IBM. Los temas fundamentales eran el desarrollo de heurísticas y el aprendizaje de máquinas. En 1957 McCarthy desarrolló el lenguaje Lisp. La IBM contrató un equipo para la investigación en esa área y el gobierno de USA aportó dinero al MIT también para investigación en 1963.

A finales de los años 50 y comienzos de la década del 60 se desarrolla un programa orientado a la lectura de oraciones en inglés y la extracción de conclusiones a partir de su interpretación, al cual su autor, Robert K. Lindsay, denomina "Sad Sam". Este podía leer oraciones del tipo "Jim es hermano de John" y "La madre de Jim es Mary", a partir de ella el sistema concluía que Mary debía ser también la madre de John. Este sistema representó un enorme paso de avance en la simulación de inteligencia humana por una máquina, pues era capaz de tomar una pieza de información, interpretarla, relacionarla con información anteriormente almacenada, analizarla y sacar conclusiones lógicas. En el mismo período de tiempo hay trabajos importantes de Herbert Gelernter, de IBM, quien desarrolla un "Demostrador Automático de Teoremas de la Geometría", Alex Bernstein desarrolla un programa para el juego de ajedrez que se considera el antecedente para "Deep Blue".

En 1961 se desarrolla SAINT (Simbolic Automatic INTegrator) por James Slagle el cual se orienta a la demostración simbólica en el área del álgebra.

En 1964 Bertrand Raphael construye el sistema SIR (Semantic Information Retrieval) el cual era capaz de comprender oraciones en inglés.
En 1970, comenzaron a aparecer los programas expertos, que predicen la probabilidad de una solución bajo un set de condiciones, entre esos proyectos

estuvo: DENDRAL, que asistía a químicos en estructuras químicas complejas euclidianas; MACSYMA, producto que asistía a ingenieros y científicos en la solución de ecuaciones matemáticas complejas.

En la década 1980, creció el uso de sistemas expertos, muchas veces diseñados para aplicaciones médicas y para problemas realmente muy complejos como MYCIN, que asistió a médicos en el diagnóstico y tratamiento de infecciones en la sangre. Otros son: R1/XCON, PIP, ABEL, CASNET, PUFF, INTERNIST/CADUCEUS, etc.

En la actualidad mucho se sigue investigando en los grandes laboratorios tecnológicos educativos y privados; sin dejar de lado los notables avances en sistemas de visión por computadora (aplicados por ejemplo, para la clasificación de artículos revueltos -tornillería o piezas marcadas por códigos de colores, por citar un caso-), control robótico autónomo (Sony, con sus robots capaces de moverse en forma casi humana y reaccionar a presiones tal como lo hace una persona al caminar), aplicaciones de lógica difusa (aplicación del tracking automático en nuestras video caseteras), etc. Sin embargo, la Inteligencia Artificial sigue en su gran mayoría acotado por su dominio tecnológico y poco ha podido salir al mercado del consumidor final o a la industria.

1.1.2. Definición de Inteligencia Artificial

Con respecto a las definiciones actuales de Inteligencia Artificial se encuentran autores como Rich & Knight [1994], Stuart [1996], quienes definen en forma general la Inteligencia Artificial como la capacidad que tienen las máquinas para realizar tareas que en el momento son realizadas por seres humanos.

Otros autores como Nebendah [1988], Delgado [1998], arrojan definiciones más completas y las definen cómo el campo de estudio que se enfoca en la explicación y emulación de la conducta inteligente en función de procesos computacionales basadas en la experiencia y el conocimiento continuo del ambiente.

Hay más autores como Marr [1977], Mompin [1987], Rolston [1992], que en sus definiciones involucran los términos de soluciones a problemas muy complejos. A criterio de los autores las definiciones de Delgado y Nebendah son muy completas, pero sin el apoyo del juicio formado, emocionalidad del ser humano pueden perder peso dichas soluciones, por eso, hay que lograr un ambiente balanceado entre ambas partes para mayor efectividad de soluciones.

La Inteligencia Artificial es un área de la investigación donde se desarrollan algoritmos para controlar cosas, y es así que en 1956 se establecen las bases para funcionar como un campo independiente de la informática.

Objetivos de la Investigación en Inteligencia Artificial

Los investigadores en Inteligencia Artificial tienen como objetivos principales:

- Reproducción automática del razonamiento humano.
- Sistemas Expertos.
- Resolución de problemas.
- Control automático.
- Bases de datos inteligentes.
- Ingeniería del software (diseños de entornos de programación inteligente).

Otros investigadores están trabajando en el reto del reconocimiento de patrones donde se espera un rápido progreso en este campo que abarca la comprensión y la síntesis del habla, el proceso de imágenes y la visión artificial.

Finalmente, la fundamental investigación sobre la representación del conocimiento, la conceptualización cognoscitiva y la comprensión del lenguaje natural.

Uno de los principales objetivos de los investigadores en inteligencia artificial es la reproducción automática del razonamiento humano.

El razonamiento de un jugador de ajedrez no siempre es el mismo que el de un directivo que se pregunta la viabilidad de fabricar un nuevo producto. Un niño jugando con bloques de madera en una mesa no tiene idea de la complejidad del razonamiento necesario para llevar a cabo la construcción de una pirámide, e intentar que un robot hiciera lo mismo que el niño requeriría un largo programa de computador.

Es por eso que existen diferentes formas de considerar situaciones complejas, las cuales mencionamos a continuación:

- Deducción, que permite obtener conclusiones de reglas cuyas premisas hemos comprobado.
- Inducción que produce reglas a partir de observaciones parciales.

Estos dos tipos principales (deducción e inducción) pueden utilizarse de un modo analítico (el razonamiento se divide en submódulos que son más difíciles de manejar, o de un modo sintético (inverso del proceso anterior, juntando elementos que se separaron anteriormente).

La inducción puede tener lugar cuando se comparan situaciones que son casi similares, con parámetros desconocidos en una situación dada asignándole los valores que tienen ya en una situación de referencia; este es un razonamiento por analogía.

1.1.3. Características de la Inteligencia Artificial

El campo de la Inteligencia Artificial se ha caracterizado por los requerimientos de procesamiento simbólico, representación del conocimiento, búsqueda en espacios de estados, y otras tareas de alto nivel de similar complejidad.

Por tal razón es importante especificar cuáles son las principales características de la Inteligencia Artificial:

1) Una característica fundamental que distingue a los métodos de Inteligencia Artificial de los métodos numéricos es el uso de símbolos no matemáticos, aunque no es suficiente para distinguirlo completamente. Otros tipos de programas como los compiladores y sistemas de bases de datos, también procesan símbolos y no se considera que usen técnicas de Inteligencia Artificial.

2) 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. El programa especifica cómo encontrar la secuencia de pasos necesarios para resolver un problema dado (programa declarativo). En contraste con los programas que no son de Inteligencia Artificial, que siguen un algoritmo definido, que especifica explícitamente cómo encontrar las variables de salida para cualquier variable dada de entrada (programa de procedimiento). Las conclusiones de un programa declarativo no son fijas y son determinadas parcialmente por las conclusiones intermedias alcanzadas durante las consideraciones al problema específico. Los lenguajes orientados al objeto comparten esta propiedad y se han caracterizado por su afinidad con la Inteligencia Artificial.

3) El razonamiento basado en el conocimiento, implica que estos programas incorporan factores y relaciones del mundo real y del ámbito del conocimiento en el que ellos operan. Al contrario de los programas para propósito específico, como los de contabilidad y cálculos científicos; los programas de Inteligencia Artificial pueden distinguir entre el programa de razonamiento o motor de inferencia y la base de conocimientos dándole la capacidad de explicar discrepancias entre ellas.

4) Aplicabilidad a datos y problemas mal estructurados, sin las técnicas de Inteligencia Artificial los programas no pueden trabajar con este tipo de problemas. Un ejemplo es la resolución de conflictos en tareas orientadas a metas como en planificación, o el diagnóstico de tareas en un sistema del mundo real: con poca información, con una solución cercana y no necesariamente exacta.

1.1.4. Prueba de Turing

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

La Prueba de Turing, en su aspecto más genérico y aceptado, se basa en que un Juez humano entable una conversación con un ser humano y una máquina (a la vez) e intente establecer cuál es la máquina. Se considera que una máquina podrá superar la prueba de Turing cuando sea capaz de comportarse tal como un humano y el juez no pueda discernir entre sus interlocutores. Para lograrlo, la máquina debería ser capaz de utilizar un lenguaje natural, razonar, tener conocimientos y aprender. Este conjunto de elementos es, en su mayoría, lo que representa obstáculos para la Inteligencia Artificial. A partir de aquí, surgen los detractores de la validez de la prueba de Turing. El argumento principal se basa en que la prueba sólo evalúa si el sujeto se parece a un ser humano y que eso no implica inteligencia.

A continuación enumeramos una serie de pasos para entender mejor su funcionamiento:

1) Dos personas y un computador, una de las personas es un interrogador y la otra persona y el computador son los elementos a ser identificados.
2) Cada uno de los elementos está en un cuarto distinto.
3) La comunicación entre los elementos es escrita y no se puede ver.
4) Después de un cierto número de preguntas y respuestas, sí el interrogador no puede identificar quién es el computador y quién la persona, entonces podemos decir que el computador piensa.

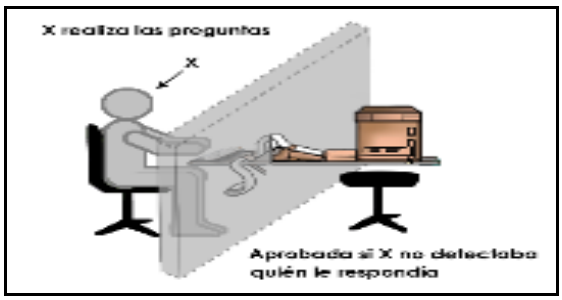


Figura 1. Prueba de Turing.

Hoy en día, el trabajo esencial de programar una computadora para pasar la prueba es considerable. La computadora debería ser capaz de lo siguiente:

- Procesar un lenguaje natural para poder establecer comunicación satisfactoria, en español, inglés o en cualquier otro idioma humano.
- Representar el conocimiento para guardar toda la información que se le proporcione antes o durante el interrogatorio. Utilización de Base de hechos para receptar preguntas y luego almacenarlas.
- Razonar automáticamente utilizando la información guardada, al responder preguntas y obtener nuevas conclusiones o tomar decisiones.
- 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: Capacidad para mover el objeto que ha sido percibido.

1.1.5. Principales ramas de la Inteligencia Artificial

La Inteligencia Artificial es una rama de la computación que se encarga, entre otras cosas, de los problemas de percepción, razonamiento y aprendizaje en sistemas artificiales, y que tienen en común diversas ramas complementarias, tales como: lógica difusa, robótica, procesamiento del lenguaje natural, sistemas de aprendizaje, sistemas expertos, redes neuronales, percepción y reconocimiento de patrones, entre otras.

Lógica Difusa

Es la rama que analiza y estructura información del mundo real en una escala entre falso y verdadero. La lógica difusa toma conceptos básicos como caliente o húmedo y les permite a los ingenieros construir televisores, acondicionadores de aire, lavadoras y otros dispositivos que juzgan información difícil de definir.

Cuando los matemáticos carecen de algoritmos para representar un sistema que debe responder a ciertas entradas, la lógica difusa es una herramienta para controlar o describir el sistema usando reglas de sentido común que se refieren a cantidades no determinadas. Los sistemas difusos frecuentemente tienen reglas tomadas de expertos; cuando no hay experto, los sistemas difusos adaptivos (adaptativos) aprenden las reglas observando cómo se manipulan sistemas reales.

Robótica

Incluye el desarrollo de dispositivos mecánicos o de computación que tengan la capacidad de realizar funciones, tales como pintar automóviles, de hacer soldaduras de precisión y realizar otras tareas que requieran de un alto grado de precisión o que sean tediosas o impliquen peligro para los seres humanos. En la robótica contemporánea se combinan las capacidades de alta precisión de la máquina con un software controlador sofisticado.

Procesamiento del lenguaje natural

Son programas diseñados para tomar lenguajes humanos como entrada y traducirlo en un conjunto estándar de instrucciones que una computadora ejecuta. Los programas analizan gramaticalmente oraciones, tratando de eliminar la ambigüedad de un contexto determinado. El propósito de estos complejos programas es permitir a los seres humanos usar su propio lenguaje natural cuando interactúan con programas como sistemas de administración de bases de datos (DBMS) o sistemas de apoyo para la toma de decisiones. El objetivo de los

procesadores de lenguaje natural es eliminar paulatinamente la necesidad de aprender lenguajes de programación o comandos personalizados para que las computadoras entiendan. Su gran ventaja radica en que pueden usarse junto con dispositivos de reconocimiento de voz a fin de que el usuario de instrucciones a las computadoras para que realicen tareas, sin usar un teclado o cualquier otro dispositivo de entrada.

Sistemas de aprendizaje

Es una combinación de software y equipos que le permite a la computadora cambiar su modo de funcionar o reaccionar frente a determinadas situaciones, basado en la retroalimentación que recibe. Por ejemplo, algunos juegos computarizados tienen capacidades de aprendizaje, si la computadora no gana un juego en particular, recuerda no hacer los mismos movimientos.

Sistemas Expertos

Son programas que reproducen el proceso intelectual de un experto humano en un campo particular, pudiendo mejorar su productividad, ahorrar tiempo y dinero, conservar sus valiosos conocimientos y difundirlos más fácilmente.

Redes neuronales

Es un sistema de computación que puede actuar en la misma forma que funciona el cerebro humano, o simularlo. Además, el software de red neuronal se puede usar para simular una red neuronal por medio de computadoras normales. Las redes neuronales pueden procesar muchas piezas de información al mismo tiempo y aprender a reconocer patrones.

Las redes neuronales son excelentes para el reconocimiento de patrones. Por ejemplo, las computadoras de red neuronal se pueden usar para leer los códigos de barra de los cheques bancarios a pesar de manchas o de una impresión de baja calidad.

Percepción y reconocimiento de patrones

Estudia la identificación, inspección, localización y verificación de patrones, Las máquinas serán capaces de reaccionar a su entorno por influencias recibidas a través de sensores y dispositivos de interacción con el exterior. Busca identificar la voz de quién está comunicándose, proveerá sonidos que simulen la voz a fin de "conversar" con el usuario. Las tecnologías del habla se encuentran en desarrollo e introducción en nuestro entorno cotidiano. El habla, medio de comunicación por excelencia entre los seres humanos, comienza a ser utilizada como medio de comunicación entre máquinas y seres humanos. Dentro de esta tecnología se engloban entre otros los procesos:

- Codificación de voz

- Síntesis de voz.
- Reconocimiento automático del habla.
- Verificación e identificación de la persona.
- Traducción automática.
- Identificación del lenguaje.

Ejemplos:

Visión y habla, reconocimiento de voz, obtención de fallos por medio de la visión, diagnósticos médicos, etc.

1.1.6. Lenguajes de programación

Los lenguajes de programación son programas que se han diseñado principalmente para simular un comportamiento inteligente, incluyen algoritmos de juego tales como el ajedrez, programas de comprensión del lenguaje natural, visión por computadora, robótica y sistemas expertos.

Estos lenguajes están basados en reglas de acción, análisis de posibilidades brindándonos la ayuda necesaria en todas las ramas de la acción humana y pueden ser conocidos como el principio de la programación en Inteligencia Artificial, ya que estos ofrecen características planteadas especialmente para manejar problemas usualmente encontrados en la Inteligencia Artificial.

Dependiendo de varios factores estos se pueden clasificar en:

1) Imperativos

Son aquellos que se basan en asignación de valores y se fundamentan en la utilización de variables para su almacenamiento y realizar operaciones con esos datos.

Ejemplos: PASCAL, C/C++.

2) Declarativos

Están basados en la definición de funciones o relaciones. No utilizan instrucciones de asignación ya que sus variables no almacenan valores. Estos lenguajes son los más fáciles de utilizar (no se requieren conocimientos específicos de informática), están muy próximos al hombre. Se suelen denominar también lenguajes de órdenes, ya que los programas están formados por sentencias que ordenan “qué es lo que se quiere hacer”, no teniendo el programador que indicar a la computadora el proceso detallado de cómo hacerlo (el algoritmo)”.

Los lenguajes declarativos se dividen en:

- **Lenguajes funcionales**

Son un tipo de lenguajes declarativos, en los que los programas están formados por una serie de definiciones de funciones matemáticas, cada una con su dominio e imagen que pueden interactuar entre ellas y combinarse mediante condicionales, recursividad y composición funcional.

Ejemplo: Lisp, el cual se suele aplicar a problemas de Inteligencia Artificial.

- **Lenguajes lógicos**

Son el otro tipo de lenguajes declarativos, y en ellos los programas están formados por una serie de definiciones de predicados. También se les denomina lenguajes de programación lógica, y el mayor exponente es el lenguaje Prolog. Se aplican sobre todo en la resolución de problemas de Inteligencia Artificial.

3) Orientados a objetos

Los objetos con operadores se comunican entre sí mediante mensajes para resolver problemas, además existen conceptos de herencia y polimorfismo.

Ejemplos: SmallTalk, Hypercard, CLOS.

1.1.6.1. Lenguajes de Inteligencia Artificial

Tradicionalmente LISP y Prolog han sido los lenguajes que se han venido utilizando para la programación de sistemas expertos a través de la Inteligencia Artificial.

Estos lenguajes ofrecen características especialmente diseñadas para manejar problemas generalmente encontrados en Inteligencia Artificial. Por este motivo se les conoce como lenguajes de inteligencia artificial.

Una de las principales características que comparten los lenguajes Lisp y Prolog, como consecuencia de su respectiva estructura, es que pueden ser utilizados para escribir programas capaces de examinar a otros programas, incluyéndose ellos mismos. Esta capacidad es útil cuando se quiere que el programa explique sus conclusiones y solo es posible cuando el programa tiene la capacidad de examinar su propio modo de operación.

LISP: Su nombre se deriva de List Processor. Lisp fue el primer lenguaje para procesamiento simbólico. John McCarthy lo desarrolló en 1958, en el Instituto de Tecnología de Massachusetts (MIT), inicialmente como un lenguaje de programación con el cual los investigadores pudieran implementar eficientemente programas de computadora capaces de razonar.

Rápidamente Lisp se hizo popular por su capacidad de manipular símbolos y fue escogido para el desarrollo de muchos sistemas de Inteligencia Artificial. Actualmente es utilizado en varios dominios que incluyen la escritura de compiladores, sistemas para diseño VLSI, sistemas para diseño mecánico asistido por computadora (AUTOCAD), animaciones gráficas y sistemas basados en conocimiento.

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

Cuando se quiere implementar un problema en este lenguaje 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 lo han usado porque posee característica como las que se muestran a continuación:

- Su forma de programación es declarativa y no procedimental.
- Posee la habilidad de expresar algoritmos recursivos que manipulen estructuras de datos dinámicos.
- Lisp es adecuado para la Inteligencia Artificial ya que el código y los datos se tratan de la misma forma (como listas), siendo sencillo escribir programas capaces de escribir otros programas.

Prolog: Es un lenguaje de programación que fue desarrollado en 1973 por Alain Colmerauer y su equipo de investigación en la Universidad de Marseille Aix. Uno de sus principales protagonistas en desarrollo y promoción, fue Robert Kowalski de la Universidad de Edimburgh. Cuyas investigaciones proporcionaron el marco teórico, mientras que los trabajos de Colmerauer dieron origen al actual lenguaje de programación, construyendo el primer interprete Prolog. David Warren, de la Universidad de Edimburgh, desarrolló el primer compilador de Prolog (WAM – “Warren Abstract Machine”), en el que se pretendía usar la lógica formal como base para un lenguaje de programación, es decir, era un primer intento de diseñar un lenguaje de programación que posibilitara al programador especificar sus problemas en lógica. Lo que lo diferencia de los demás es el énfasis sobre la especificación del problema. Inicialmente fue utilizado para el procesamiento de lenguaje natural, pero posteriormente se popularizó entre los desarrolladores de aplicaciones de Inteligencia Artificial por su capacidad de manipulación simbólica. Prolog es una realización aproximada del modelo de computación de Programación Lógica sobre una máquina secuencial. Desde luego, no es la única realización posible, pero sí es la mejor elección práctica, ya que equilibra por un lado la preservación de las propiedades del modelo abstracto de Programación Lógica y por el otro lado consigue que la implementación sea eficiente.

A partir de 1981 tuvo una importante difusión en todo el mundo, especialmente porque los japoneses decidieron utilizarlo en el desarrollo de sus sistemas de computación de la quinta generación para que fueran Sistemas de Procesamiento de Conocimiento.

Prolog es un lenguaje utilizado para implementar Inteligencia Artificial y Sistemas Expertos. Gran parte de su éxito se debe a su conveniencia por ser código abierto (modificable) y se obtiene fácilmente en internet, además de su capacidad de deducción de respuestas para las consultas realizadas, Prolog es un lenguaje simple y fácil de programar, hasta para principiantes, pero sus motores de inferencia no siempre son eficientes. Sus aplicaciones varían desde sistemas ambientales hasta la resolución de funciones automatizadas.

Una de las características que hacen de Prolog un lenguaje de gran interés es su reducido número de mecanismos de soporte, entre los que se encuentran el reconocimiento de patrones, la unificación, el reintento ("backtracking"), el manejo de listas y de estructuras de datos flexibles, así como la recursividad.

Adicionalmente, tal como lo ofrece el lenguaje de programación Lisp, Prolog incluye estructuras de listas dinámicas que le dan gran poderío y flexibilidad; en ellas se puede hacer referencia explícita a un elemento de una lista o al resto de ella (cola). Por ejemplo:

```
miembro(Elemento,[Elemento | _]).
miembro(Elemento,[ _ |Lista]):-miembro(Elemento,Lista).
```

La regla verifica si elemento es parte de una lista o no.

Prolog permite definiciones recursivas (p.ej., como la de miembro) y estructuras de datos que, al igual que en el caso de Lisp, corresponden a una notación uniforme con los programas; lo que permite que un programa se modifique a sí mismo durante el proceso de ejecución y que genere como resultado un comportamiento no determinístico, es decir que no siempre genere los mismos resultados ante las mismas entradas.

Entre sus características más importantes tenemos:

- Esta basado en lógica y programación declarativa.
- No se especifica cómo debe hacerse, sino que debe lograrse.
- Una característica importante en Prolog que lo diferencia de otros lenguajes de programación, es que una variable solo puede tener un valor mientras se cumple el objetivo.
- El programador se concentra más en el conocimiento que en los algoritmos.
- En Prolog, se llega a una solución infiriéndola desde algo ya conocido.

Típicamente, un programa en Prolog no es una secuencia de acciones, sino una colección de hechos que junto con reglas permiten obtener soluciones o llegar a conclusiones utilizando los hechos ya establecidos.

OPS5: Official Production System 5 (OPS5), es un lenguaje para ingeniería cognoscitiva que soporta el método de representación del conocimiento en forma de reglas. Incorpora un módulo unificador, un intérprete que incluye un mecanismo

de encadenamiento progresivo, y herramientas para edición y depuración de los programas.

OPS5 es un miembro de la familia de lenguajes de programación desarrollados en la Universidad Carnegie - Mellon. Varias compañías han desarrollado implementaciones comerciales de OPS5, para diferentes plataformas.

1.1.6.2. Ventajas y desventajas entre Prolog y Lisp

Ventajas de Prolog

- Una ventaja desde el punto de vista del usuario es la facilidad para programar ya que se pueden escribir programas rápidamente, con pocos errores originando programas claramente legibles, aun si no se conoce muy bien el lenguaje.

- Los elementos básicos utilizados por el lenguaje Prolog son las fórmulas atómicas. Los términos son denominados objetos en el lenguaje, por lo tanto, se tendrán constantes simbólicas para nombrar a los predicados, constantes variables (símbolos alfanuméricos que comiencen con mayúsculas) y funciones para denotar los objetos.

- En prolog se utiliza notación prefija e infija.

- Un programa en Prolog se conforma con dos tipos de expresiones (llamadas "cláusulas"): "hechos" y "reglas".

- Un programa en Prolog tiene como estructura básica la base de hechos, ella define relaciones siempre verdaderas.

- Prolog utiliza el Manejo dinámico y automático de memoria.

- Prolog utiliza un mecanismo de búsqueda independiente de la base de hechos. Aunque pueda parecer algo retorcido, es una buena estrategia puesto que garantiza el proceso de todas las posibilidades. Es útil para el programador conocer dicho mecanismo a la hora de depurar y optimizar los programas.

- Prolog es un lenguaje lógico es por eso que la habilidad de Prolog para calcular de forma procedural es una de las ventajas específicas que tiene. Como consecuencia esto anima al programador a considerar el significado declarativo de los programas de forma relativamente independiente de su significado procedural. Es decir, las ventajas de la forma declarativa de este lenguaje son claras (es más fácil pensar las soluciones y muchos detalles procedurales son resueltos automáticamente por el propio lenguaje) y podemos aprovecharlas.

- No hay que pensar demasiado en la solución del problema, ya que Prolog infiere sus respuestas basándose en las reglas declaradas dentro del programa.

- Modularidad: cada predicado (procedimiento) puede ser ejecutado, validado y examinado independiente e individualmente. Prolog no tiene variables globales, ni asignación. Cada relación está auto contenida, lo que permite una mayor modularidad, portabilidad y reusabilidad de relaciones entre programas.

- Polimorfismo: se trata de un lenguaje de programación sin tipos, lo que permite un alto nivel de abstracción e independencia de los datos (objetos).

- En Prolog, se puede representar incluso los mismos programas como estructuras.

- La ejecución y búsqueda incorporada en el lenguaje Prolog, lo hace por medio de una descripción válida de un problema a través del cual obtiene automáticamente una conclusión efectiva.

- El orden de ejecución de las instrucciones no tiene nada que ver con el orden en que fueron escritas. Tampoco hay instrucciones de control propiamente dichas. Para trabajar con este lenguaje, un programador debe acostumbrarse a pensar de una manera muy diferente a la que se utiliza en los lenguajes clásicos.

- Los comentarios en prolog van antecedidos por el signo % y permite comentar varias líneas poniendo siempre al final el mismo signo, lo hace como en el lenguaje C.

- Prolog puede funcionar en modo interpretado, por lo que podemos interactuar con el ejecutando programas desde su prompt.

Ventajas de Lisp

- Lisp es difícil de leer y depurar por todos los paréntesis que requiere.

- Lisp usa una sintaxis basada totalmente en paréntesis ya que son necesarios para delimitar el ámbito de aplicabilidad de una operación.

- Lisp solo utiliza únicamente notación prefija.

- En Lisp los programas se construyen a partir de la composición de funciones.

- Lisp depende de una implementación de memoria virtual.

- Los procedimientos y datos en Lisp tienen la misma forma. Un programa en Lisp puede usar otro programa como dato, incluso puede crear otro programa y usarlo, simplifica la escritura de programas en Lisp que transformen y manipulen a otros programas en este lenguaje.

- Lisp es un lenguaje funcional. Esto quiere decir que todas las operaciones a realizar se consideran funciones que devuelven un resultado siempre.

- Lisp usa mecanismos tomados del cálculo lambda para definir nombres de funciones (abstracción).

- Las instrucciones se ejecutan normalmente en orden secuencial, es decir, una a continuación de otra, en el mismo orden en que están escritas, que sólo varía cuando se alcanza una instrucción de control (un bucle, una instrucción condicional o una transferencia).

- Los comentarios en Lisp son de una línea y precedidos por el punto y coma. No se diferencian mayúsculas de minúsculas en los nombres de funciones, variables y etc.

En la siguiente tabla mostramos las diferencias y semejanzas entre Prolog y Lisp:

Prolog	Lisp
programas claramente legibles	Difícil de leer y depurar
Sintaxis basada en hechos y reglas.	Sintaxis basada en paréntesis
Notación prefija e infija	notación prefija
Estructura básica la base de hechos	Composición de funciones con listas
Manejo dinámico y automático de memoria	implementación de memoria virtual
Modularidad de relaciones entre programas	Los procedimientos y datos en Lisp tienen la misma forma
Prolog es un lenguaje lógico	Lisp es un lenguaje funcional
Programación sin tipos	Usa mecanismos tomados del cálculo lambda
Orden secuencial, variando cuando hay que descifrar alguna variable	Ejecución en orden secuencial
Las mayúsculas y minúsculas tiene un significado distinto.	No se diferencian mayúsculas ni minúsculas en variables

Tabla 1. Diferencia y semejanza entre Prolog y Lisp

Desventajas:

- La resolución automática no siempre es eficiente, por lo que eventualmente se podría dar una respuesta incorrecta a una consulta.

- Ciertos problemas están ligados a la representación del conocimiento, que Prolog no posee.

- Prolog algunas veces es incapaz de reconocer que un problema es (para su propio conocimiento) inaplicable o insuficiente. Si el programa no contiene suficiente información para contestar una consulta, es incapaz de reconocerlo y responde no. En esta situación sería más eficiente conocer que la respuesta no es negativa, sino que no es posible inferir un resultado.

- Los motores de inferencia poseen algunos límites Prolog esta limitado a un mecanismo de retroseguimento (bactraking), esto genera objetivos y subobjetivos a conseguir pero no dispone de medios para determinar las mejores reglas a seleccionar.

1.1.7. Sistemas de desarrollo

Históricamente, los primeros sistemas basados en conocimiento fueron desarrollados utilizando lenguajes de programación como Lisp y Prolog A medida que el desarrollo de sistemas basados en conocimiento iba aumentado en cantidad y complejidad, la comunidad científica comenzó a buscar formas de desarrollar los sistemas en menor tiempo y con menor esfuerzo.

Posteriormente ingresaron al mercado otras herramientas que incorporaron, además de opciones de representación del conocimiento, esquemas de inferencia y control. Estas herramientas tomaron el nombre de entornos de desarrollo de sistemas basados en conocimiento.

A continuación mencionamos algunos ejemplos de sistemas comerciales:

- **Sistemas vacíos (shells):** EMYCIN, Crystal, Leonardo, XiPlus, EXSYS, VP-Expert, Intelligence Compiler.

- **Entornos híbridos de desarrollo:** CLIPS, KEE, ART, EGERIA, Kappa, Nexpert Object, Goldworks, LOOPS, Flavors.

Es importante mencionar que el trabajo que estamos elaborando esta basado en el lenguaje de programación Prolog del cual detallaremos más adelante su funcionamiento

1.1.8. Ciencias que aportan a la Inteligencia Artificial

Como ocurre casi siempre en el caso de una ciencia recién creada, la Inteligencia Artificial 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 Inteligencia Artificial 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 utilizando 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.

1.2. Introducción a los Sistemas Expertos

1.2.1. Definición de los Sistemas Expertos

Un Sistema Experto es básicamente un programa de computadora basado en conocimientos y razonamientos que lleva a cabo tareas que generalmente sólo realiza un experto humano. Es un programa que imita el comportamiento humano porque utiliza la información que se le proporciona para poder dar una opinión sobre un tema en especial. Los Sistemas Expertos también permiten resolver problemas específicos de manera inteligente y satisfactoria. La tarea principal de un Sistema Experto es tratar de aconsejar al usuario ya que son útiles para resolver problemas que se basan en conocimiento.

1.2.2. Antecedentes de los Sistemas Expertos

Los Sistemas Expertos surgen a mediados de los años sesenta. Los primeros investigadores que desarrollaron programas basados en leyes de razonamiento fueron Alan Newell y Herbert Simón, quienes desarrollaron el GPS. Este sistema era capaz de resolver problemas como el de las torres de Hanoi y otros similares, a través de la criptoaritmética. Sin embargo, este programa no podía resolver problemas más “cotidianos” y reales, como, por ejemplo, dar un diagnóstico médico.

Entonces algunos investigadores cambiaron el enfoque del problema y se dedicaban a resolver problemas sobre un área específica intentando simular el

razonamiento humano. Ya no computarizaban la inteligencia general, se centraron en dominios de conocimiento muy concretos. De esta manera nacieron los Sistemas Expertos.

El primer Sistema Experto que se aplicó a problemas más reales fue desarrollado en 1965 con el fin de identificar estructuras químicas, el programa se llamó DENDRAL. Lo que este Sistema Experto hacía, al igual que lo hacían los expertos de entonces, era tomar unas hipótesis relevantes como soluciones posibles, y someterlas a prueba comparándolas con los datos. El nombre DENDRAL significa árbol en griego. Debido a esto, el programa fue bautizado así porque su principal tarea era buscar en un árbol de posibilidades la estructura del compuesto.

El siguiente Sistema Experto que causó gran impacto fue el MYCIN, en 1972, era una aplicación que detectaba trastornos en la sangre y recetaba los medicamentos requeridos. Fue tal el éxito de MYCIN que incluso se llegó a utilizar en algunos hospitales. Para 1973, se creó Tieresias, cuya función era la de servir de intérprete o interfaz entre los especialistas que manejaban MYCIN cuando introducían nuevos conocimientos. Tieresias entraba en acción cuando MYCIN cometía un error en un diagnóstico, por la falta de información o por alguna falla en el árbol de desarrollo de teorías, corrigiendo la regla que generaba el resultado o bien destruyéndola. Para 1980 se implantó en la DEC (Digital Equipment Corporation) el primer Sistema Experto, el XCON. Para esto se tuvieron que dedicar dos años al desarrollo de este Sistema Experto. Y aunque en su primer intento al implantarse en 1979 consiguió sólo el 20% del 95% de la resolución de las configuraciones de todas las computadoras que salieron de la DEC, volvió al laboratorio de desarrollo otro año más, y a su regreso le resultó en un ahorro de 40 millones de dólares a dicha compañía, lo cual fue un gran beneficio para dicha compañía.

Para los años comprendidos entre 1980 y 1985 se crearon diversos Sistemas Expertos, tales como el DELTA de la General Electric Company, el cual se encargaba de reparar locomotoras diesel y eléctricas, o como “Aldo en Disco”, que reparaba calderas hidrostáticas giratorias usadas para la eliminación de bacterias.

En esa misma época surgen empresas dedicadas a desarrollar Sistemas Expertos, las cuales en conjunto invirtieron un total de más de 300 millones de dólares. Los productos más importantes que creaban estas nuevas compañías eran las "máquinas Lisp", las cuales consistían en unos ordenadores que ejecutaban programas Lisp con la misma rapidez que en un ordenador central.

1.2.3. Estructura básica de un Sistema Experto

Un Sistema Experto esta formado básicamente de:

Base de conocimientos

Es la parte del Sistema Experto que contiene el conocimiento sobre el dominio. Hay que obtener el conocimiento del experto y codificarlo en la base de

conocimientos. Una forma clásica de representar el conocimiento en un Sistema Experto son las reglas. Una regla es una estructura condicional que relaciona lógicamente la información contenida en la parte del antecedente con otra información contenida en la parte del consecuente.

Base de hechos (Memoria de trabajo)

Contiene los hechos sobre un problema que se ha descubierto durante una consulta. Con el Sistema Experto, el usuario introduce la información del problema actual en la base de hechos. El sistema empareja esta información con el conocimiento disponible en la base de conocimientos para deducir nuevos hechos.

Motor de inferencia

Es el corazón de todo Sistema Experto la misión principal de este componente es la obtención de conclusiones mediante la aplicación del conocimiento abstracto al conocimiento concreto. En el transcurso de este proceso, si el conocimiento inicial es muy limitado, y el sistema no puede obtener ninguna conclusión, se utilizará el subsistema de explicación. Con este módulo el Sistema Experto modela el proceso de razonamiento humano. Dicho motor trabaja con la información contenida en la base de hechos para deducir nuevos hechos y así obtener conclusiones acerca del problema.

Subsistema de explicación

Una característica de los Sistemas Expertos es su habilidad para explicar su razonamiento. Usando este módulo, un sistema experto puede dar información al usuario de por qué está haciendo una pregunta y cómo ha llegado a una conclusión. Este módulo brinda beneficios tanto al programador del sistema como al usuario. El programador puede usarlo para detectar errores y el usuario se beneficia de la transparencia del sistema.

Interfaz de usuario

La interacción entre un Sistema Experto y un usuario se realiza en lenguaje natural. También es altamente interactiva y se asemeja a la conversación entre seres humanos. Para que este proceso sea aceptable para el usuario es muy importante el diseño de interfaz de usuario. Un requerimiento básico de la interfaz es la habilidad de hacer preguntas. Para obtener información fiable del usuario debemos ser cuidadosos en su diseño, lo que requiere diseñar la interfaz usando menús o gráficos.

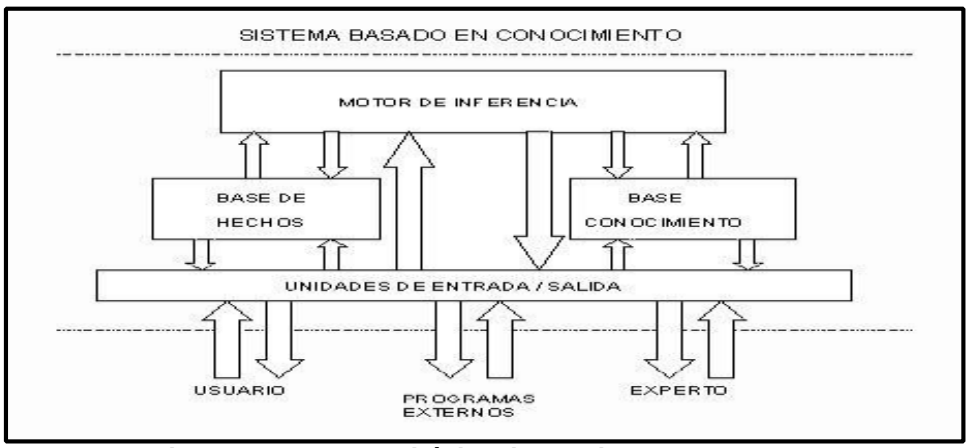


Figura 2. Estructura básica de un Sistema Experto.

1.2.4. Tipos de Sistemas Expertos

Principalmente existen tres tipos de sistemas expertos:

1) Basados en reglas

Este tipo de sistemas trabajan mediante la aplicación de reglas y la comparación de resultados. También pueden trabajar por inferencia lógica dirigida, bien empezando con una evidencia inicial en una determinada situación y dirigiéndose hacia la obtención de una solución, o bien con hipótesis sobre las posibles soluciones y volviendo hacia atrás para encontrar una evidencia existente (o una deducción de una evidencia existente) que apoye una hipótesis en particular.

2) Basados en casos o CBR (Case Based Reasoning)

El Razonamiento basado en casos fue formalizado en cuatro pasos con el propósito de ser utilizado en razonamiento de computadora.

Recordar

Dado un determinado problema, recordar los casos relevantes que pueden solucionarlo. Un caso consiste en un problema, una solución y típicamente anotaciones sobre como la solución fue llevada a acabo. Por ejemplo, supongamos que Ramiro quiere preparar crepes (panqueques) de dulce de leche siendo un cocinero novato. Y la experiencia más relevante que él puede recordar es aquella en la cual tuvo éxito una vez preparando unos crepes con crema. El conocimiento de Ramiro está basado en el procedimiento que utilizó para hacer los crepes correctamente, en conjunto con las decisiones que él haya tomado en la elaboración de los mismos.

Reutilizar

Adaptar la solución del problema anterior a este nuevo. En el ejemplo anterior, Ramiro deberá adaptar el problema, sacando la crema y reemplazándola por dulce de leche.

Revisar

Una vez adaptado el problema probar la solución en el mundo real o en una simulación y si es necesario revisarla. Continuando con el ejemplo anterior supongamos que Ramiro rellena con dulce el crepe una vez que está servido pero el dulce no se esparce bien y enfría el crepe, siendo este un efecto no deseado. Al revisar nuevamente encuentra como solución colocar el dulce de leche cuando el crepe aún está en la sartén, así se calentará y esparcirá mejor.

Retener

Después de que la solución ha sido adaptada satisfactoriamente para solucionar el problema dado, almacenar la experiencia resultante como un nuevo caso en la memoria. En nuestro ejemplo Ramiro almacena en su memoria esta experiencia enriqueciendo el conjunto de casos, de esta manera hará cada vez mejor los crepes.

3) Basados en redes bayesianas

Una red bayesiana es un modelo probabilístico multivariado que relaciona un conjunto de variables aleatorias mediante un grafo. Gracias a su motor de actualización de probabilidades, las redes bayesianas son una herramienta extremadamente útil en la estimación de probabilidades ante nuevas evidencias.

En cada uno de los tipos de Sistemas Expertos mencionados anteriormente, la solución a un problema planteado se obtiene de la siguiente forma:

- Aplicando reglas heurísticas apoyadas generalmente en lógica difusa para su evaluación y aplicación.

- Aplicando el razonamiento basado en casos, donde la solución a un problema similar planteado con anterioridad se adapta al nuevo problema.

- Aplicando redes bayesianas, basadas en estadística y en el teorema de Bayes.

1.2.5. Ventajas y Limitaciones de los Sistemas Expertos

- **Ventajas**

Permanencia: A diferencia de un experto humano un Sistema Experto no envejece, y por tanto no sufre pérdida de facultades con el paso del tiempo.

Duplicación: Una vez programado un Sistema Experto lo podemos duplicar infinidad de veces.

Rapidez: Un Sistema Experto 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 Sistema Experto puede trabajar en entornos peligrosos o dañinos para el ser humano.

Fiabilidad: Los Sistemas Expertos no se ven afectados por condiciones externas, un humano sí (cansancio, presión, etc.).

- **Limitaciones**

Sentido común: Para un Sistema Experto no hay nada obvio. Por ejemplo, un Sistema Experto sobre medicina podría admitir que un hombre lleva 40 meses embarazado, si no se le especifica que esto no es posible.

Lenguaje natural: Con un experto humano podemos mantener una conversación informal mientras que con un Sistema Experto no podemos.

Capacidad de aprendizaje: Cualquier persona aprende con relativa facilidad de sus errores y de errores ajenos, que un Sistema Experto 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 Sistema Experto 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 Sistema Experto no es capaz de manejar conocimiento estructurado.

1.2.6. Tareas que realiza un Sistema Experto

Monitorización

Es un caso particular de la interpretación, y consiste en la comparación continua de los valores de las señales o datos de entrada y unos valores que actúan como criterios de normalidad o estándares. En el campo del mantenimiento predictivo los Sistemas Expertos se utilizan fundamentalmente como herramientas de

diagnóstico. Se trata de que el programa pueda determinar en cada momento el estado de funcionamiento de sistemas complejos, anticipándose a los posibles incidentes que pudieran acontecer. Así, usando un modelo computacional del razonamiento de un experto humano, proporciona los mismos resultados que alcanzaría dicho experto.

Diseño

Es el proceso de especificar una descripción de un artefacto que satisface varias características desde un número de fuentes de conocimiento.

El diseño se concibe de dos formas:

1) El diseño en ingeniería es el uso de principios científicos, información técnica e imaginación en la definición de una estructura mecánica, máquina o sistema que ejecute funciones específicas con el máximo de economía y eficiencia.

2) El diseño industrial busca rectificar las omisiones de la ingeniería, es un intento consciente de traer forma y orden visual a la ingeniería de hardware donde la tecnología no provee estas características.

Los Sistemas Expertos en diseño ven este proceso como un problema de búsqueda de una solución óptima o adecuada. Las soluciones alternas pueden ser conocidas de antemano o se pueden generar automáticamente probándose distintos diseños para verificar cuáles de ellos cumplen los requerimientos solicitados por el usuario, ésta técnica es llamada “generación y prueba”, por lo tanto estos Sistemas Expertos son llamados de selección. En áreas de aplicación, la prueba se termina cuando se encuentra la primera solución; sin embargo, existen problemas más complejos en los que el objetivo es encontrar la solución más óptima.

Planificación

Es la realización de planes o secuencias de acciones y es un caso particular de la simulación. Está compuesto por un simulador y un sistema de control. El efecto final es la ordenación de un conjunto de acciones con el fin de conseguir un objetivo global.

Los problemas que presenta la planificación mediante Sistemas Expertos son los siguientes:

- Existen consecuencias no previsibles, de forma que hay que explorar y explicar varios planes.

- Existen muchas consideraciones que deben ser valoradas o incluirles un factor de peso.

- Suelen existir interacciones entre planes de subobjetivos diversos, por lo que deben elegirse soluciones de compromiso.

- Trabajo frecuente con incertidumbre, pues la mayoría de los datos con los que se trabaja son más o menos probables pero no seguros.

- Es necesario hacer uso de fuentes diversas tales como bases de datos.

Control

Un sistema de control participa en la realización de las tareas de interpretación, diagnóstico y reparación de forma secuencial. Con ello se consigue conducir o guiar un proceso o sistema. Los sistemas de control son complejos debido al número de funciones que deben manejar y el gran número de factores que deben considerar; esta complejidad creciente es otra de las razones que apuntan al uso del conocimiento, y por tanto de los Sistemas Expertos.

Los sistemas de control pueden ser de dos tipos:

Lazo abierto. Si en el mismo la realimentación o el paso de un proceso a otro lo realiza el operador.

Lazo cerrado. Si no tiene que intervenir el operador en ninguna parte del mismo.

Reparación, corrección, terapia o tratamiento

Consiste en la proposición de las acciones correctoras necesarias para la resolución de un problema. Los Sistemas Expertos en reparación tienen que cumplir diversos objetivos, tales como:

- Reparación lo más rápida y económicamente posible.

- Orden de las reparaciones cuando hay que realizar varias.

- Evitar los efectos secundarios de la reparación, es decir la aparición de nuevas averías por la reparación.

Simulación

Es una técnica que consiste en crear modelos basados en hechos, observaciones e interpretaciones sobre la computadora, a fin de estudiar el comportamiento de los mismos mediante la observación de las salidas para un conjunto de entradas. Las técnicas tradicionales de simulación requieren modelos matemáticos y lógicos que describen el comportamiento del sistema bajo estudio. El empleo de los Sistemas Expertos para la simulación viene motivado por la principal característica de los mismos, que es su capacidad para la simulación del comportamiento de un experto humano, que es un proceso complejo.

En la aplicación de los Sistemas Expertos para simulación hay que diferenciar cinco configuraciones posibles:

1) Un Sistema Experto puede disponer de un simulador con el fin de comprobar las soluciones y en su caso rectificar el proceso que sigue.

2) Un sistema de simulación puede contener como parte del mismo a un Sistema Experto y por lo tanto el Sistema Experto no tiene que ser necesariamente de simulación.

3) Un Sistema Experto puede controlar un proceso de simulación, es decir que el modelo está en la base de conocimiento del Sistema Experto y su evolución es función de la base de hechos, la base de conocimientos y el motor de inferencia, y no de un conjunto de ecuaciones aritmético – lógicas.

4) Un Sistema Experto puede utilizarse como consejero del usuario y del sistema de simulación.

5) Un Sistema Expertos puede utilizarse como máscara o sistema frontal de un simulador con el fin de que el usuario reciba explicación y justificación de los procesos.

Instrucción

Un sistema de instrucción realizará un seguimiento del proceso de aprendizaje. El sistema detecta errores ya sea de una persona con conocimientos e identifica el remedio adecuado, es decir, desarrolla un plan de enseñanza que facilita el proceso de aprendizaje y la corrección de errores.

1.2.7. Ejemplos de Sistemas Expertos

1) **DENDRAL:** (Interpreta la estructura molecular) Fue el primer Sistema Experto en ser utilizado para propósitos reales, al margen de la investigación computacional, y durante aproximadamente 10 años, el sistema tuvo cierto éxito entre químicos y biólogos, ya que facilitaba enormemente la inferencia de estructuras moleculares.

2) **MYCIN:** Es un Sistema Experto desarrollado a principios de los años 70 por Edgar ShortLiffe, en la Universidad de Stanford. Fue escrito en LIPS, e inicialmente estaba inspirado en DENDRAL. Su principal función consistía en el diagnóstico de enfermedades infecciosas de la sangre; además, MYCIN era capaz de “razonar” el proceso seguido para llegar a estos diagnósticos, y de recetar medicaciones personalizadas a cada paciente (según su estatura, peso, etc.).

3) **XCON:** Es un Sistema Experto para configuraciones, desarrollado por la Digital Equipment Corporation. Según los deseos individuales del cliente se configuran

redes de ordenadores VAX. Ya que el abanico de productos que se ofrecen en el mercado es muy amplio, la configuración completa y correcta de un sistema de estas características es un problema de gran complejidad. Responde esencialmente a dos preguntas: ¿Pueden conjugarse los componentes solicitados por el cliente de forma conveniente y razonable? Y ¿Los componentes de sistema especificados son compatibles y completos? Las respuestas a estas preguntas son muy detalladas. XCON es capaz de comprobar y completar los pedidos entrantes mucho más rápido y mejor que las personas encargadas de hacerlo antes que él.

Tema 2: Introducción al lenguaje de programación Prolog.

En este tema se presentan las principales características del lenguaje, como fundamento para analizar su potencialidad de acuerdo con diversos enfoques educativos, así como el manejo de los distintos comandos que presenta el lenguaje de programación Prolog para el desarrollo de bases de hechos y sus respectivas consultas.

La programación lógica es un paradigma de los lenguajes de programación en el cual los programas se consideran como una serie de aserciones lógicas. De esta forma, el conocimiento se representa mediante reglas, tratándose de sistemas declarativos. Una representación declarativa es aquélla en la que el conocimiento está especificado, pero en la que la manera en que dicho conocimiento debe ser usado no viene dado. El más popular de los sistemas de programación lógica es Prolog.

Contenido:

2. Introducción al lenguaje de programación Prolog.
2.1. Introducción a Prolog.
2.2. Historia.
2.3. Descripción.
2.4. Predicados en Prolog.
2.4.1. Variables.
2.4.1.1 Cómo se asignan valores a las variables
2.4.1.2 Variables anónimas.
2.4.2. Reglas.
2.4.3. Términos en Prolog.
2.5. Estructura de un programa Prolog.
2.6. Preguntas.
2.7.1. El mecanismo de unificación.
2.7. Tipos de datos en Prolog.
2.8.1. Operadores
2.8. Estructura de datos.
2.9. Recursividad.
2.10.1. Definición de recursividad
2.10. Entrada y salida de datos.
2.11.1. Lectura y escritura de términos.
2.11.2. Escritura con formato
2.11.3. Lectura y escritura de caracteres
2.11. Desarrollo de Versiones Actuales
2.12. Ejercicios resueltos.

2. Introducción al lenguaje de programación Prolog

2.1. Introducción a Prolog

Prolog es un lenguaje de programación diseñado para representar y utilizar el conocimiento que se tiene sobre un determinado dominio. Los programas en Prolog responden preguntas sobre el tema del cual tienen conocimiento.

Prolog es un lenguaje de programación especialmente indicado para modelar problemas que impliquen objetos y las relaciones entre ellos. Está basado en los siguientes mecanismos básicos: unificación, estructuras de datos basadas en árboles y backtracking automático. La sintaxis del lenguaje incluye la declaración de hechos, preguntas y reglas. Con la definición de este pequeño conjunto de conceptos se consigue un lenguaje de programación muy potente y flexible, ampliamente utilizado (junto con el lenguaje de programación Lisp) en aplicaciones que utilizan técnicas de Inteligencia Artificial.

La popularidad de este lenguaje se debe a su capacidad de deducción y además es un lenguaje fácil de usar por su semántica y sintaxis. Sólo busca relaciones entre los objetos creados, las variables y las listas, que son su estructura básica.

2.2. Historia

Los inicios de la programación lógica se dan gracias a los primeros trabajos de inteligencia artificial. Los cuales originaron el primer lenguaje de programación que contempla los mecanismos de inferencia necesarios para la demostración automática de teoremas.

El lenguaje de programación Prolog se originó del trabajo hecho por Robert A. Kowalski en la Universidad de Edinburgh y Alain Colmeraur en la Universidad de Aix-Marseille (Francia) en los años 70. La investigación de Kowolski en el área de deducción automatizada, llevó al desarrollo con Colmerauer al uso formal de lógica como un lenguaje de programación. Kowolski proporcionó la base teórica y Colmerauer inició la programación de Prolog. Colmeraur y Phillipe Roussel desarrollaron el primer intérprete, y David Warren de la Universidad de Edinburgh desarrolló el primer compilador Prolog. La mayoría de las implementaciones comerciales de Prolog usan la misma sintaxis desarrollada en Edinburgh. Su nombre proviene de las palabras en ingles “Programming in Logic”, que significa programación lógica.

2.3. Descripción

Prolog es un lenguaje de programación simple, pero poderoso. Se basa en nociones matemáticas de relaciones de inferencia. Es un lenguaje declarativo e interpretado, esto quiere decir que el lenguaje se usa para representar conocimientos sobre un determinado dominio y las relaciones entre objetos de ese dominio.

Un programa en Prolog consiste en una base de hechos de relaciones lógicas y detalles que se cumplen para la aplicación. Dicha base no tiene una estructura impuesta, ni un procedimiento o clase principal. Los datos y relaciones de un programa en Prolog se escriben en un único archivo, el cual es consultado por el programa cuando se le hace una pregunta.

Escribir un programa en Prolog consiste en declarar el conocimiento disponible acerca de los objetivos, además de sus relaciones y sus reglas. En lugar de tener que abrir un programa para ejecutar cierta aplicación y obtener una solución, se hace una pregunta, el programa revisa la base de hechos para encontrar la solución a la pregunta. Si existe más de una solución, Prolog hace backtracking para encontrar soluciones distintas. El propio sistema es el que deduce las respuestas a las preguntas que se le plantean, dichas respuestas las deduce del conocimiento obtenido por el conjunto de reglas dadas.

La ejecución de Prolog consiste en una búsqueda en profundidad de un árbol conteniendo todas las posibles soluciones. Para cada una de ellas se evaluará su validez. La estructura de un programa en Prolog es lógica y directa. Se explican cada una de sus partes y operadores disponibles a continuación.

2.4. Predicados en Prolog

Los predicados son los elementos ejecutables en Prolog. En muchos sentidos se asemejan a los procedimientos o funciones típicos de los lenguajes imperativos. Una llamada concreta a un predicado, con unos argumentos concretos, se denomina objetivo (en inglés, goal). Todos los objetivos tiene un resultado de éxito o fallo tras su ejecución, indicando si el predicado es cierto para los argumentos dados, o por el contrario, es falso.

Cuando un objetivo tiene éxito, las variables libres que aparecen en los argumentos pueden quedar ligadas. Estos son los valores que hacen cierto el predicado. Si el predicado falla, no ocurren ligaduras en las variables libres.

Se utilizan para expresar propiedades de los predicados, y relaciones entre ellos, en Prolog los llamaremos hechos.

Debemos tener en cuenta que:

- Los nombres de todos los objetos y relaciones deben comenzar con una letra minúscula.
- Primero se escribe la relación o propiedad: predicado.
- Y los objetos se escriben separándolos mediante comas y encerrados entre paréntesis: argumentos.
- Al final del hecho debe ir un punto (".").

2.4.1. Variables

Las variables se utilizan para representar objetos cualesquiera del universo u objetos desconocidos en ese momento, es decir, son las incógnitas del problema. Se diferencian de los átomos en que empiezan siempre con una letra mayúscula o con el signo de subrayado (_). Así, deberemos ir con cuidado ya que cualquier identificador que empiece por mayúscula, será tomado por Prolog como una variable. En una consulta simple, las variables nos pueden servir para que Prolog encuentre un dato.

Las variables en Prolog pueden ser instanciadas o no-instanciadas. Una variable es instanciada cuando existe algún objeto que se pueda corresponder a la variable (lo que está buscando). Es no-instanciada cuando no se encuentra lo que se está buscando.

Cuando a Prolog se le hace una pregunta que contiene una variable, este busca por todos los hechos para encontrar un objeto en el cual instanciarse.

Ejemplo:

```
mcd(X,X,X).
mcd(X,Y,D):-X<Y,Y1 is Y-X.
mcd(X,Y,D):-Y<X, mcd(Y,X,D).

?- mcd(100,10,X).
X= 10
Yes
```

2.4.1.1. Cómo se asignan valores a las variables

Como habrás podido observar, en Prolog no existe una instrucción de asignación. Esto resulta una de las diferencias fundamentales entre Prolog y el resto de lenguajes de programación.

Las variables en Prolog toman valores al ser “igualadas” a constantes en los hechos o reglas del programa. Hasta el momento en que una variable toma un valor, se dice que está desinstanciada; cuando ha tomado un valor, se dice que está instanciada (a dicho valor). Sin embargo, una variable solo permanece instanciada hasta el momento en que obtenemos una solución. Después, se desinstancia, y se procede a buscar nuevas soluciones.

2.4.2. Variables anónimas

La mayoría de los programas lógicos son susceptibles de incluir en algunas de sus cláusulas variables que son irrelevantes para el proceso de resolución, pero cuya presencia implica una complicación innecesaria de tal proceso. Como muestra tomemos el siguiente ejemplo:

Supóngase que, dado el predicado factorial, se necesita definir a partir de él un predicado **es _ factorial(X)** que es cierto si X es el factorial de algún número natural. Una forma de definir este predicado sería estableciendo que X es un factorial siempre y cuando exista algún Y tal que X sea el factorial de ese Y, es decir:

es _ factorial(X):- factorial (Y, X).

Como podemos observa en este caso el posible valor de la variable Y es indiferente, puesto que aparece una sola vez en la estructura de sus respectivas cláusulas y por tanto se trata de variables irrelevantes a nivel del proceso de unificación, puesto que no son incluidas en ninguna de las ecuaciones consideradas. Sin embargo, una vez introducidas en la cláusula, el intérprete lógico no siempre puede detectar su irrelevancia. Como consecuencia, son asimiladas en el proceso de resolución como una variable mas, sujeta a renombramientos, inclusión en las sustituciones. En definitiva, las variables en cuestión solo consiguen dificultar innecesariamente el trabajo del intérprete, puesto que desde un punto de vista lógico su funcionalidad es nula.

Con el objeto de evitar este tipo de situaciones, la mayoría de los intérpretes Prolog incluyen el concepto de variable anónima, denotada **"_".** Este tipo de variable, simplemente es ignorada en el proceso de resolución.

Así que volviendo a nuestro ejemplo, podríamos reescribir el programa anterior en la forma:

es _ factorial(X):- factorial (_, X).

2.4.3. Reglas

Las reglas se utilizan en Prolog para significar que un hecho depende de uno ó más hechos. Son la representación de las implicaciones lógicas del tipo p ---> q (p implica q). Una regla consiste en una cabeza y un cuerpo, unidos por el signo ":-". La cabeza está formada por un único hecho, el cuerpo puede ser uno ó mas hechos (conjunción de hechos), separados por una coma (","), que actúa como el "y" lógico y todas finalizan con un punto (".").

La cabeza en una regla Prolog corresponde al consecuente de una implicación lógica, y el cuerpo al antecedente. Este hecho puede conducir a errores de representación.

Ejemplo:

Representación lógica	Representación Prolog
es_impar(X) ----> verif(X ,número)	verif(X, número):- es_impar(X).

2.4.4. Términos en Prolog

Los términos son el único elemento del lenguaje, es decir, los datos son términos, el código son términos, incluso el propio programa es un término. No obstante, es habitual llamar término solamente a los datos que maneja un programa.

Un término se compone de un átomo seguido de cero a N argumentos entre paréntesis y separados por comas. Los números enteros o decimales sin restricciones de tamaño también son términos.

Los términos pueden ser constantes o variables, y suponemos definido un dominio no vacío en el cual toman valores (Universo del Discurso).

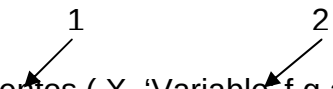
Las constantes: Se utilizan para dar nombre a objetos concretos del dominio, dicho de otra manera, representan tipos especificos conocidos de nuestro universo. Además, las constantes atómicas de Prolog también se utilizan para representar propiedades y relaciones entre los objetos del dominio. Hay dos clases de constantes:

1) Átomos

Existen tres clases de constantes atómicas:

- Cadenas de letras, dígitos y subrayado (_) empezando por letra minúscula.
- Cualquier cadena de caracteres encerrada entre comillas simples (').
- Combinaciones especiales de signos: "?-", ":-",
  Algunos ejemplos: X Respuesta _x1 _12

Ejemplo:



muchos _ argumentos (X ,'Variable',f,g,a).

?- escribecadena(L):-name(L,Lascii),

2) Números

Se utilizan para representar números de forma que se puedan realizar operaciones aritméticas.

- **Enteros:** en la implementación de Prolog puede utilizarse cualquier entero en el intervalo [-223,223-1]= [-8.388.608,8.388.607].

  Ejemplos: 1 3212 0 -223
  El entero es el que mas se usa en Prolog.

- **Reales:** decimales en coma flotante, consistentes en al menos un dígito, opcionalmente un punto decimal y más dígitos, un «+» o «-» y más dígitos.

Ejemplos: 3.14 -0.0216 100.2

2.5. Estructura de un programa

El hecho de programar en Prolog consiste en dar al ordenador un universo finito en forma de hechos y reglas, proporcionando los medios para realizar inferencias de un hecho a otro. A continuación, si se hacen las preguntas adecuadas, Prolog buscará las respuestas en dicho universo y las presentará en la pantalla. La programación en Prolog consiste en:

- Declarar algunos HECHOS sobre los objetos y sus relaciones.
- Definir algunas REGLAS sobre los objetos y sus relaciones, y
- Hacer PREGUNTAS sobre los objetos y sus relaciones.

Programa Prolog: Conjunto de afirmaciones (hechos y reglas) representando los conocimientos que poseemos en un determinado dominio o campo de nuestra competencia. Un sistema Prolog está basado en un comprobador de teoremas por resolución para cláusulas de horn. La regla de resolución no nos dice que cláusulas elegir ni que términos unificar dentro de cada cláusula. La estrategia de resolución particular que utiliza Prolog es una forma de resolución de entrada lineal (árbol de búsqueda estándar). Para la búsqueda de cláusulas alternativas para satisfacer el mismo objetivo, Prolog adopta una estrategia de primero hacia abajo (recorrido del árbol en profundidad). Por todo esto, el orden de las cláusulas (hechos y reglas) de un determinado procedimiento es importante en Prolog, ya que determina el orden en que las soluciones serán encontradas, e incluso puede conducir a fallos en el programa. Más importante es, si cabe, el orden de las metas a alcanzar dentro del cuerpo de una regla.

Una cláusula de Horn es una regla de inferencia lógica con una serie de premisas (cero, una o más), y un único consecuente. Las cláusulas de Horn son las instrucciones básicas del lenguaje de programación Prolog.
La sintaxis de una cláusula de Horn en PROLOG tiene el siguiente aspecto: dividir(X, Y):-par(X), suma(X, Y).

Ejecución del programa: Demostración de un teorema en este universo, es decir, demostración de que una conclusión se deduce de las premisas (afirmaciones previas).

Programa Prolog = Base de Conocimientos + Motor de Inferencia.

2.6. Preguntas

Las preguntas son las herramientas que tenemos para recuperar la información desde Prolog. Al hacer una pregunta a un programa lógico queremos determinar si

esa pregunta es consecuencia lógica del programa. Prolog considera que todo lo que hay en la base de hechos es verdad, y lo que no, es falso. Cuando se hace una pregunta a Prolog, éste efectuará una búsqueda por toda la Base de hechos intentando encontrar hechos que coincidan con la consulta. Dos hechos “coinciden” (se pueden unificar) si sus predicados son el mismo (se escriben de igual forma) y si cada uno de los respectivos argumentos son iguales entre sí.

2.6.1. El mecanismo de unificación

La unificación es el mecanismo mediante el cual las variables lógicas toman valor en Prolog. El valor que puede tomar una variable consiste en cualquier término, por ejemplo, j(3), 23.2, ’hola que tal’, etc. Por eso decimos que los datos que maneja Prolog son términos. Cuando una variable no tiene valor se dice que está libre. Pero una vez que se le asigna valor, éste ya no cambia, por eso se dice que la variable está ocupada.

Se dice que dos términos unifican cuando existe una posible ligadura (asignación de valor) de las variables, tal que ambos términos son idénticos sustituyendo las variables por dichos valores. Por ejemplo: a(X, 3) y a (4, Z) unifican dando valores a las variables: X vale 4, Z vale 3. Obsérvese que las variables de ambos términos entran en juego.

Por otra parte, no todas las variables están obligadas a quedar ligadas. Por ejemplo: h(X) y h (Y) unifican aunque las variables X e Y no quedan ligadas. No obstante, ambas variables permanecen unificadas entre sí. Si posteriormente ligamos X al valor j (3) (por ejemplo), entonces automáticamente la variable Y tomará ese mismo valor. Lo que está ocurriendo es que, al unificar los términos dados, se impone la restricción de que X e Y deben tomar el mismo valor aunque en ese preciso instante no se conozca dicho valor.

La unificación no debe confundirse con la asignación de los lenguajes imperativos, puesto que representa la igualdad lógica. Para saber si dos términos unifican podemos aplicar las siguientes normas:

1) Una variable siempre unifica con un término, quedando ésta ligada a dicho término.

2) Dos variables siempre unifican entre sí. Además, cuando una de ellas se liga a un término, todas las que unifican se ligan a dicho término.

3) Para que dos términos unifiquen, deben tener el mismo functor y la misma aridad. Después se comprueba que los argumentos unifican uno a uno manteniendo las ligaduras que se produzcan en cada uno.

4) Si dos términos no unifican, ninguna variable queda ligada.

2.7. Tipos de datos en Prolog

Prolog no es un lenguaje con asignación de tipos fuerte, la lógica se preocupa más de las relaciones entre hechos que del tipo de éstos, dando a todos ellos un tratamiento similar.

2.7.1. Operadores

Son predicados predefinidos en Prolog para las operaciones matemáticas básicas. Su sintaxis depende de la posición que ocupen, pudiendo ser infijos ó prefijos. Por ejemplo el operador suma ("+"), podemos encontrarlo en forma prefija '+(2,5)' ó bien infija, '2 + 5'. También disponemos de predicados de igualdad y desigualdad.

Al igual que en otros lenguajes de programación es necesario tener en cuenta la precedencia y la asociatividad de los operadores antes de trabajar con ellos.

En cuanto a precedencia, es la típica. Por ejemplo, 3+2*6 se evalúa como 3+(2*6). En lo referente a la asociatividad, Prolog es asociativo por la izquierda. Así, 8/4/4 se interpreta como (8/4)/4. De igual forma, 5+8/2/2 significa 5+((8/2)/2).

La precedencia de los operadores más utilizados es:

Operador	Símbolo	Precedencia	Especificador
Potencia	^ ó **	200	xfx
Producto	*	400	yfx
División	/	400	yfx
División entera	//	400	yfx
Resto división entera	mod	400	yfx
Suma	+	500	yfx
Signo positivo	+	500	fx
Resta	-	500	yfx
Signo negativo	-	500	fx
Igualdad	=	700	xfx
Distinto	\=	700	xfx
Menor que	<	700	xfx
Menor o igual que	=<	700	xfx
Mayor que	>	700	xfx
Mayor o igual que	>=	700	xfx
Evaluación aritmética	is	700	xfx

Tabla 2. Precedencia de los operadores.

Precedencia es un entero indicando la clase de precedencia, especificador es un átomo indicando la posición y la asociatividad, y nombre es un átomo indicando el nombre que queremos que tenga el operador.

Para la posición-asociatividad utilizamos átomos especiales del tipo:

xfx xfy yfy yfx xf yf fx fy

que nos ayudan a ver el uso del posible operador, representando f al operador, x un argumento indicando que cualquier operador del argumento debe tener una clase de precedencia estrictamente menor que este operador, e y un argumento indicando que puede contener operadores de la misma clase de precedencia o menor.

Ejemplo:

El operador - declarado como yfx determina que la expresión a - b - c sea interpretada como (a - b) - c, y no como a - (b - c) ya que la x tras la f exige que el argumento que va tras el primer - contenga un operador de precedencia estrictamente menor:

7 - 3 – 2 (7 - 3) - 2 = 4 - 2 = 2 correcta

El predicado de igualdad está predefinido. Cuando se intenta satisfacer el objetivo. Goal: X=Y Prolog intenta hacer coincidir X e Y, y el objetivo se satisface si ambas coinciden.

Las reglas para decidir si X e Y son iguales son las siguientes:

1) Si X es una variable no instanciada e Y está instanciada a un valor, entonces X e Y son iguales. Además, X quedará instanciada a lo que valga Y.
2) Si X e Y son variables no instanciadas, el objetivo se satisface, y las dos variables quedan compartidas. Si dos variables quedan compartidas, en el momento en que una de ellas quede instanciada a un término, la otra queda automáticamente instanciada al mismo término.
3) Las constantes son siempre iguales a sí mismas.
4) Dos hechos son iguales si tienen el mismo nombre y el mismo número de argumentos, y todos y cada uno de los correspondientes argumentos son iguales.

Nota: Mayor información ver apartado anexos.

2.8. Estructuras de datos

Una estructura es un único objeto que se compone de una colección de otros objetos, llamados componentes, lo que facilita su tratamiento. Una estructura se escribe en Prolog especificando su nombre, y sus componentes (argumentos). Los

componentes están encerrados entre paréntesis y separados por comas; el nombre se escribe justo antes de abrir el paréntesis:

nombre (comp1, comp2,..., compn).

No disponer de tipos de datos no implica dejar de lado a las estructuras de datos, las estructuras de datos recursivas se integran naturalmente a los programas escritos en Prolog. Una estructura de datos se puede decir recursiva si cada una de sus distintas instancias satisfacen: Ser una instancia básica, la cual puede ser definida de forma directa, ser una instancia general, la cual se define como una elaboración de otra u otras instancias más simples.

A continuación mostramos un ejemplo práctico para entenderle mejor:

Introducimos una estructura de datos fecha (Dia, Mes, Año) que representa la fecha. Primero necesitamos un constructor de la estructura de datos fecha que hace la estructura de datos de día, mes y año:

hacer_fecha (D, M, A, fecha (D, M, A)).

Segundo, definimos las funciones para acceder a los componentes de la estructura de datos de la siguiente forma:

obtener_año (fecha (_, _, A), A).
obtener _ mes (fecha (_, M, _), M).
obtener _ día (fecha (D, _, _), D).

Obtener_xxx puede ser usado para probar o generar el componente correspondiente de la estructura de datos, pero no puede ser usado para colocar el valor de la componente. Así, tenemos que definir colocar_xxx para colocar valores a los componentes de la estructura de datos fecha.

colocar_año(A, fecha (D, M, _), fecha (D, M, A)).
colocar_mes (M, fecha (D, _, A), fecha (D, M, A)).
colocar _ día(D,fecha(_,M,A),fecha(D,M,A)).

2.9. Recursividad en Prolog

El área de la programación es muy amplia y con muchos detalles. Los programadores necesitan ser capaces de resolver todos los problemas que se les presente a través del ordenador aún cuando el lenguaje que utilizan no encuentre una manera directa de resolver los problemas. En el lenguaje de programación Prolog, se puede aplicar una técnica que se le di el nombre de recursividad por su funcionalidad. Esta técnica es utilizada en la programación para resolver problemas que tengan que ver con el factorial de un número, o juegos de lógica.

2.9.1. Definición de recursividad

La recursividad en el lenguaje Prolog se define como reglas que se llaman así mismas para poder realizar la misma operación o cálculos muchas veces.

Para poder implementar las reglas debemos de tener en cuenta que se necesita de una función inicial, que nos ayudará a que el bucle no se vuelva infinito.

Por lo general, cuando se tiene definiciones recursivas, es necesario considerar dos posibles casos:

1) **Caso básico**. Es el momento en el que se detiene la recursión.
2) **Caso Recursivo**. Suponiendo que ya se ha solucionado un caso más simple, cómo descomponer el caso actual hasta llegar al caso simple.

Tanto el caso básico como el caso recursivo no tienen porqué ser únicos. Como ejemplo útil se puede presentar la definición recursiva del factorial de un número:

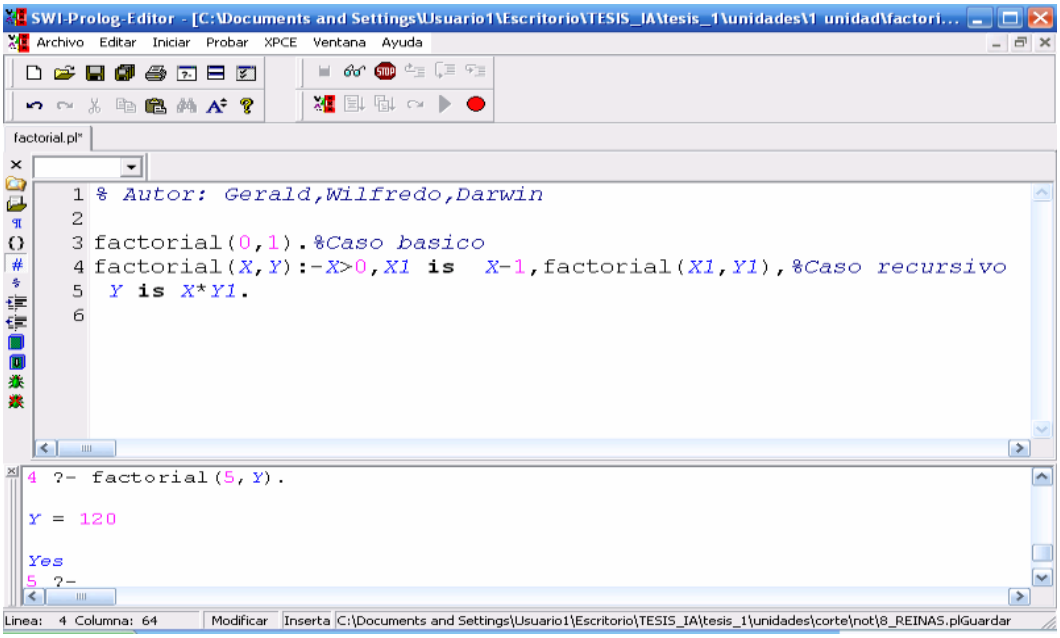
$$\text{factorial}(n) = \begin{cases} 1 & n=0 \\ n * \text{factorial}(n-1) & n>0 \end{cases}$$

En donde el factorial de 0 es por definición 1 y los factoriales de números mayores se calculan mediante la multiplicación de 1 * 2 *…., incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando el factorial.

Ejemplo para calcular recursivamente el factorial de 5:

```
factorial (5)= 5* factorial (4) ; caso recursivo
factorial (4)= 4* factorial (3) ; caso recursivo
factorial (3)= 3* factorial (2) ; caso recursivo
factorial (2)= 2* factorial (1) ; caso recursivo
factorial (1)= 1* factorial (0) ; caso recursivo
factorial (0)= 1                ; caso base
```

Solución:



2.10. Entrada y salida en Prolog

En Prolog la Entrada/Salida se realiza mediante efectos laterales sobre sus canales activos. Los predicados de E/S tienen un escaso valor lógico, pero, al ser encontrados por el sistema, ejecutan las acciones correspondientes. El principal cambio producido al estandarizar el lenguaje Prolog ha sido la redefinición de los predicados clásicos de E/S. Sin embargo, muchas implementaciones continúan ofreciendo el repertorio anterior. Los predicados de Entrada/Salida, para mantener la consistencia del sistema tienen que cumplir que:

1) Se evalúan siempre a verdad
2) Nunca se pueden resatisfacer: la reevaluación continua hacia la izquierda
3) Tienen un efecto lateral (efecto no lógico durante su ejecución): entrada o salida de un carácter, término.

2.10.1. Lectura y escritura de términos

Write

Escribe el término sobre el canal de salida activo. Su sintaxis es write (+Termino) en donde todas las variables no instanciadas de término se escriben como variables nuevas precedidas por _. Se tienen en cuenta las declaraciones de los operadores para imprimirlos con su formato adecuado (prefijo, infijo o posfijo) dentro del término.

Ejemplo:

saludo:- write (’Bienvenido a mi programa’).

Las comillas simples encierran constantes, mientras que todo lo que se encuentra entre comillas dobles es tratado como una lista.
write(X).

Display

Es idéntico a write pero sin tener en cuenta las declaraciones de operadores (todos se consideran prefijos).

nl

El predicado nl escribe un retorno de carro y una nueva línea en la salida.

Ejemplo:

write(’linea 1’), nl, write(’linea 2’). Tiene como resultado:
linea 1
linea 2

2.10.2. Escritura con formato

tab(+X)

Desplaza el cursor a la derecha X espacios. X debe estar instanciada a un entero (una expresión evaluada como un entero positivo).

read (-Termino)

Lee el siguiente término que se teclee desde el ordenador. Debe ir seguido de un punto “.” y un “retorno de carro”. Instancia Término al término leído, si Termino no estaba instanciada. Si Termino si que está instanciada, los comparará y se satisfará o fracasará dependiendo del éxito de la comparación.

2.10.3. Lectura y escritura de caracteres

El carácter es la unidad más pequeña que se puede leer y escribir. Prolog trata a los caracteres en forma de enteros correspondientes a su código ASCII.

Put

Escribe el carácter sobre el canal de salida. Su sintaxis es **put (+Carácter).** Si el carácter está instanciada a un entero entre 0...255, saca por pantalla el carácter

correspondiente a ese código ASCII y si carácter no está instanciada o no es entero, se produce un error y falla.

get(-Caracter)

Lee caracteres desde el teclado, instanciando carácter al primer carácter imprimible que se teclee. Si carácter ya está instanciada, los comparará satisfaciéndose o fracasando.

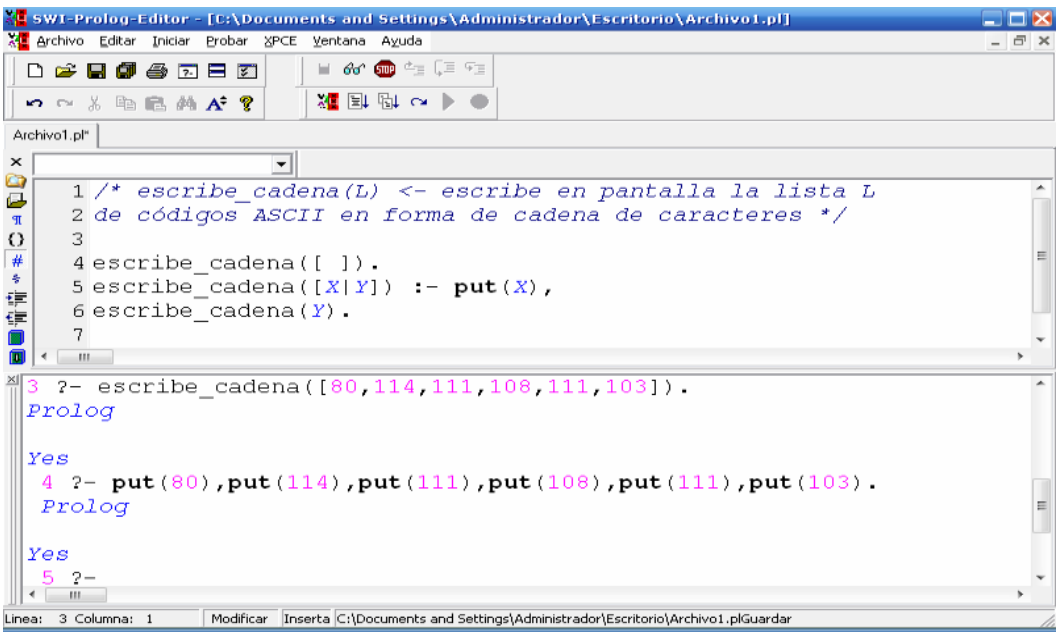
get0(-Caracter)

Igual que el anterior, sin importarle el tipo de carácter tecleado.

skip(+Caracter)

Lee del teclado hasta el carácter Caracter o el final del fichero.

Ejemplo del uso de estos predicados:



2.11. Desarrollo de versiones actuales

Antes de que existieran sistemas expertos, sistemas inteligentes adaptables o cualquier otro tipo de programa capaz de funcionar con Inteligencia Artificial, se necesito crear los lenguajes para desarrollarlo. Para ello, se consideraron algunos requerimientos básicos como la posibilidad de procesar símbolos de todo tipo y la capacidad de hacer inferencias asociadas con el lenguaje, todo dentro de un ambiente flexible que permitiera escribir el programa de forma interactiva. Uno de los lenguajes que más éxito ha tenido es Prolog. Por tal razón son muchas las compañías de software que han creado sus propias versiones del mismo. La

diferencia es mínima entre versiones ya que su sintaxis y semántica es la misma, la variación que más resalta son el cambio de plataforma para el cual fueron desarrollados.

Prolog1- Esta versión la ofrece Expert Systems International, se utiliza en maquinas que trabajan con los sistemas operativos MS-DOS, CP/M-86, RSX-11M y RT-11. Su mayor fortaleza radica en su intérprete Prolog86, este intérprete contiene un editor de cláusulas del sistema. Tiene un mejor manejo de los tipos de datos entero y real, y además posee más predicados integrados por lo cual el programador ya no requiere definirlos.

Prolog QUINTUS- Es una versión avanzada del lenguaje. El objetivo de sus diseñadores era producir una versión que pudiera ofrecer velocidades rápidas de ejecución, así como la capacidad de comunicación con otros programas. Esta versión funciona en los sistemas operativos UNIX y VMS. Una de las características interesantes es la interfaces al editor EMACS, esto provocara que la pantalla se parta en dos ventanas, en una estará el código del archivo fuente en la parte superior, mientras prolog correrá en la parte inferior, esto brinda una ayuda ya que cualquier cambio en las cláusulas del archivo, podría ser probada inmediatamente, solamente interactuando entre ventanas.

MACPROLOG- Esta versión esta diseñada totalmente para correr en máquinas MAC.Esta versión combina sofisticadas técnicas de programación de inteligencia artificial en ambientes Macintosh. Al integrar el compilador Prolog con las herramientas de MAC (ratón, menú, ventanas y gráficos), Prolog ofrece un entorno de programación sofisticado que antes solo se podía obtener con hardware costoso y muy especializado.

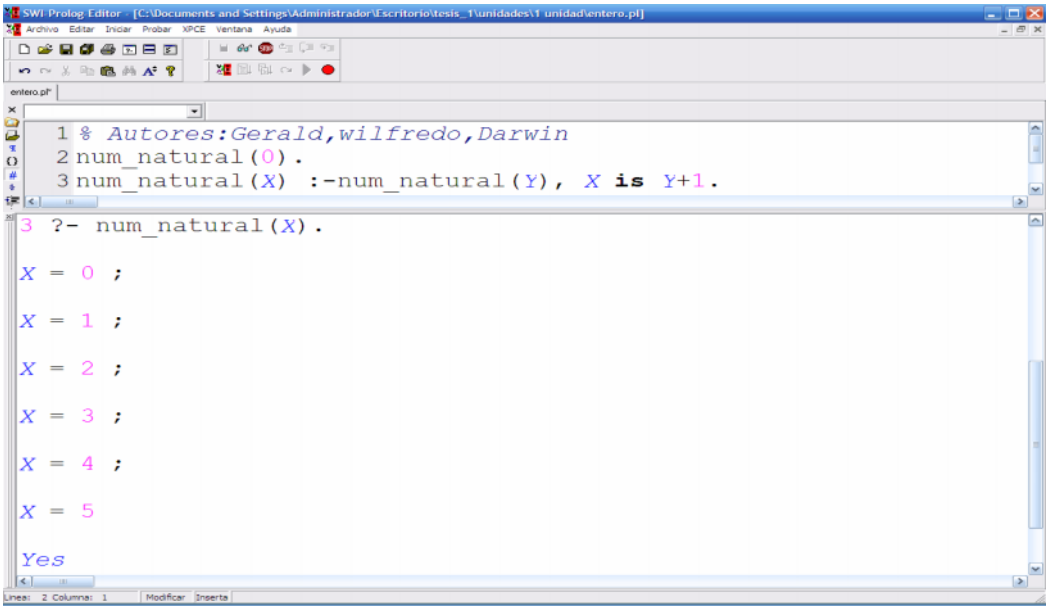
SWI-Prolog-Editor: Es una implementación de Prolog basada en un subconjunto del WAM (Warren Abstract Machine). SWI-Prolog-Editor ha sido diseñado e implementado de tal modo que puede ser empleado fácilmente para experimentar con la lógica de la programación y con las relaciones que esta mantiene con otros paradigmas de la programación, tales como el entorno PCE orientado al objeto. Posee un rico conjunto de predicados incorporados que hacen posible el desarrollo de aplicaciones robustas. Además ofrece una interfaz para el lenguaje C. Trabajaremos con SWI-Prolog-Editor (versión 5.6.48) para nuestro soporte “Componente curricular de Inteligencia Artificial y sistemas expertos”.

2.12. Ejercicios resueltos

1) Definir la relación número natural(X) que permita verificar si X es un número natural.

Nota: Recordemos que el cero es un número natural y cualquier otro número lo es, si es el sucesor de uno de ellos.

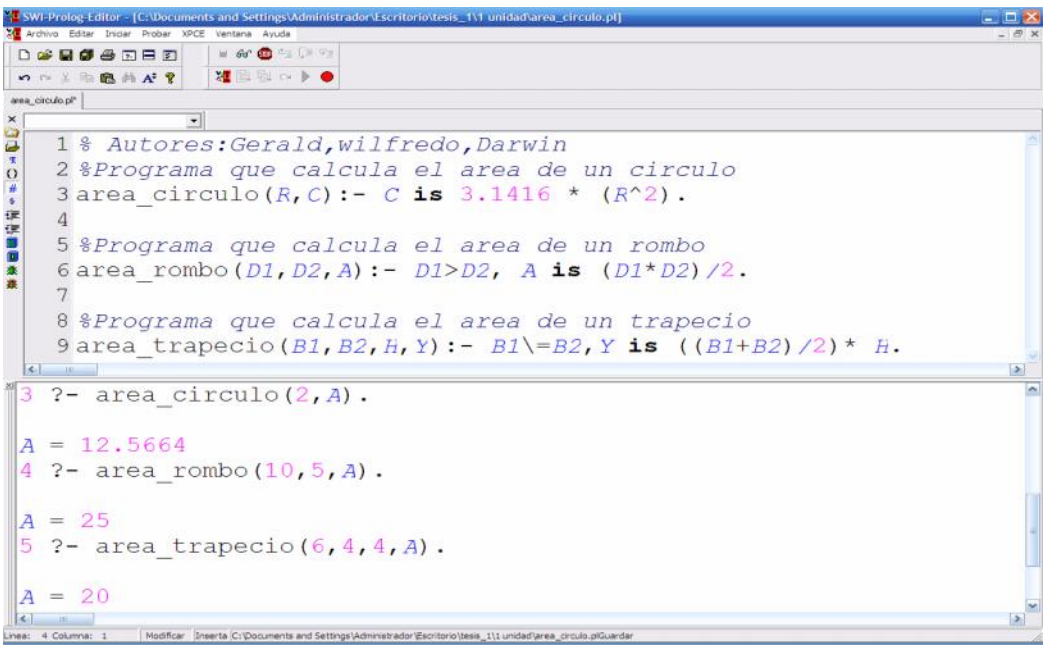
Solución:



2) Escriba un programa Prolog que permita realizar las siguientes operaciones:

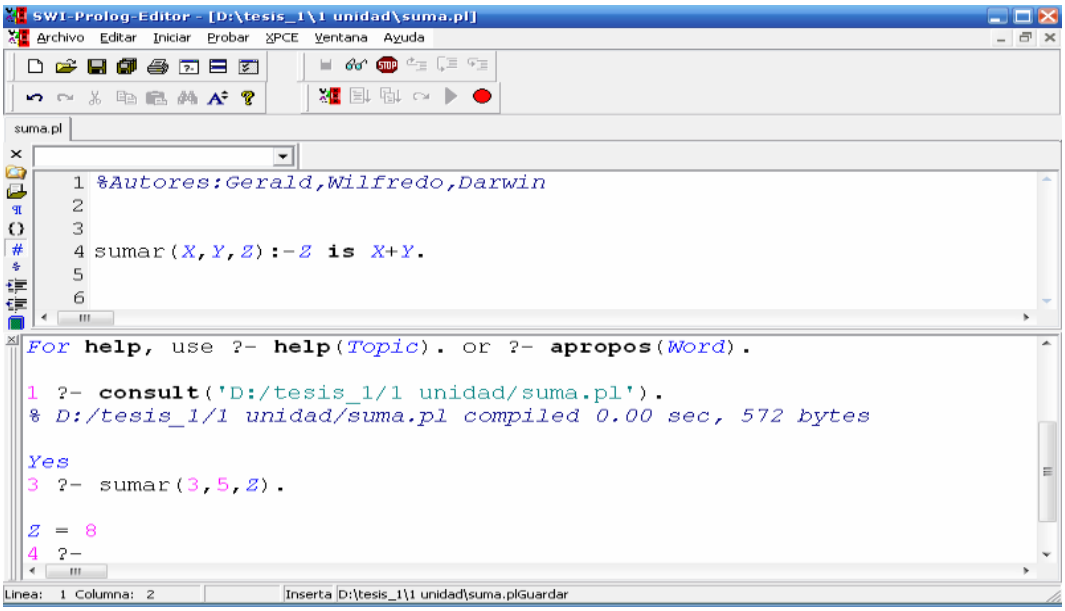
a) Área de un círculo dado el radio.
b) Área de un rombo dadas las diagonales mayor y menor.
c) Área de un trapecio dados la base mayor y menor y la altura.

Solución:



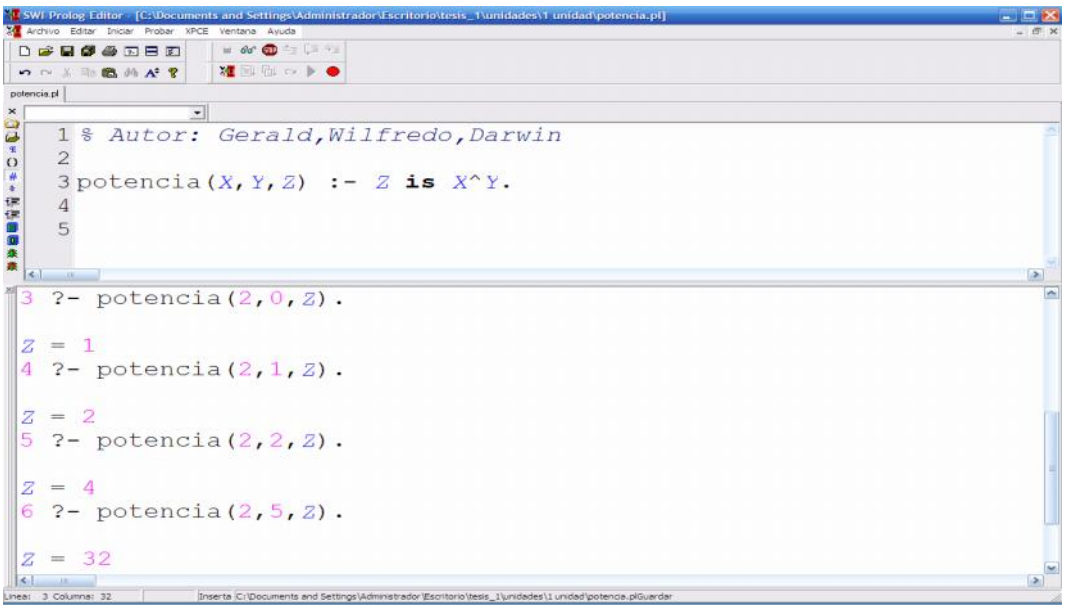
3) Definir la relación sumar(X, Y, Z) que permita verificar si Z es el resultado de la suma entre X e Y.

Solución:



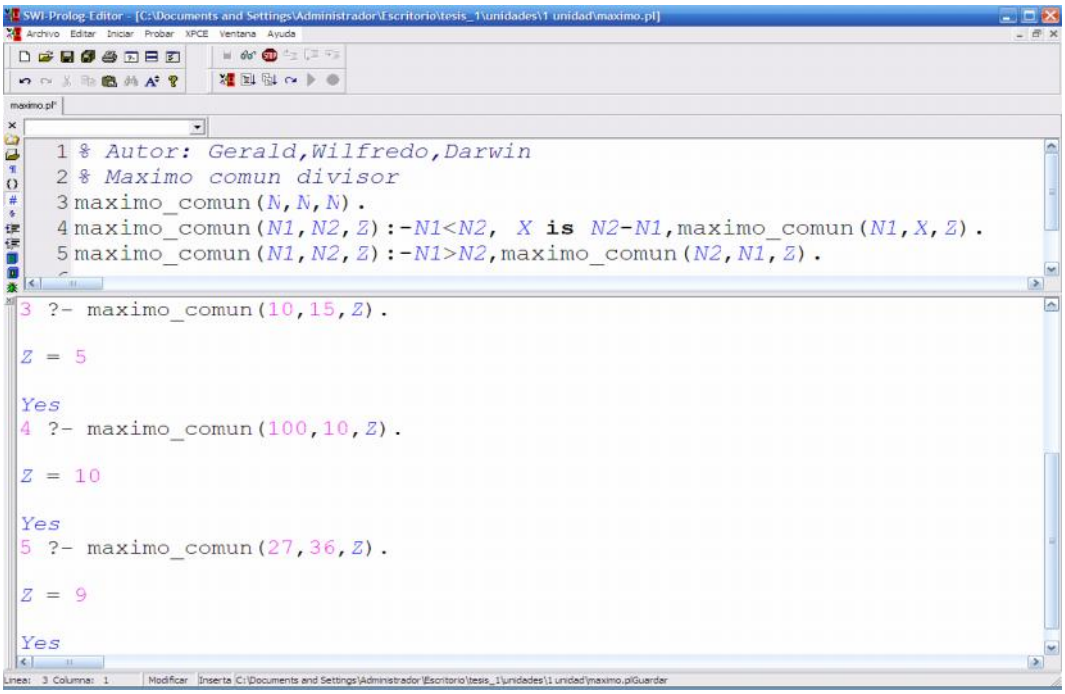
4) Definir la relación potencia(X, Y, Z) que permita verificar si Z es igual a X elevado a Y.

Solución:



5) Definir la relación máximo _común (N1, N2, Z) que permita verificar si Z es el máximo común divisor de N1 e N2.

Solución:



Tema 3: Listas en Prolog

En este tema se pretende dar a conocer al alumno el concepto de lista y el funcionamiento básico de las operaciones realizadas con ellas, las cuales serán estudiadas a través de diversos ejemplos que nos ayuden a comprender su uso.

Contenido:

3.1. Definición de lista.
3.2. Estructura con lista.
3.2.1. Simbología
3.3. Operaciones sobre lista.
3.3.1. Construcción de lista.
3.3.2. Inserción de un elemento.
3.3.3. Primer elemento.
3.3.4. Ultimo elemento.
3.3.5. Penúltimo elemento.
3.3.6. Resto de una lista.
3.3.7. Relación de pertenencia.
3.3.8. Selección de un elemento.
3.3.9. Concatenación de lista.
3.3.10. Lista inversa.
3.3.11. Palíndromo.
3.3.12. Permutación.
3.3.13. Rotación de un elemento.
3.3.14. Sublista.
3.3.15. Subconjunto de una lista.
3.3.16. Lista con todos los elementos iguales.
3.3.17. Paridad de la longitud de una lista.
3.4. Aritmética con lista.
3.4.1. Suma.
3.4.2. Producto.
3.4.3. Longitud de lista.
3.5. Ejercicios resueltos.

3. Listas en Prolog

3.1. Definición de lista

Las listas son estructuras de datos muy comunes en la programación no numérica, ya que son secuencias ordenadas de elementos que puede tener cualquier longitud. En Prolog, una lista es un objeto que contiene un número arbitrario de otros objetos, cada uno de los cuales se conocen como elementos de la lista. Estos elementos pueden ser cualquier término (constantes, variables, estructuras) los cuales se representa como una serie de elementos separados por comas y encerrados entre corchetes.

3.2. Estructura de una lista

Una lista se puede definir recursivamente como:

1) Una lista vacía [ ] sin elementos.
2) Una estructura con dos componentes:

Cabeza: Primer argumento, es el primer elemento de la lista y puede ser un átomo o una lista.
Cola: Segundo argumento, es decir, el resto de los elementos de la lista.

3.2.1. Simbología

Para hacer uso práctico de la capacidad de dividir las listas en cabeza y cola, Prolog proporciona una notación especial con la que se definen las listas en los programas:

El corchete abierto/cerrado: El cual se usa para denotar el inicio y el final de una lista.
El separador: Su símbolo es | y es usado para permitir que una lista sea representada como una cabeza y una cola.

Ejemplos:

Lista	Cabeza (elemento)	Cola (lista)
[alicia,gerald,nubia]	alicia	[gerald,nubia]
[a]	a	[ ]
[ ]	(no tiene)	(no tiene)
[[los, niños], juegan]	[los,niños]	[juegan]
[jorge,[uriel,juan], amigos]	jorge	[[uriel,juan],amigos]
[darwin, canta]	darwin	[canta]

Tabla 3. Ejemplos de Lista en Prolog.

3.3. Operaciones con lista

3.3.1. Construcción de listas

Consiste en añadir un nuevo elemento a la lista.

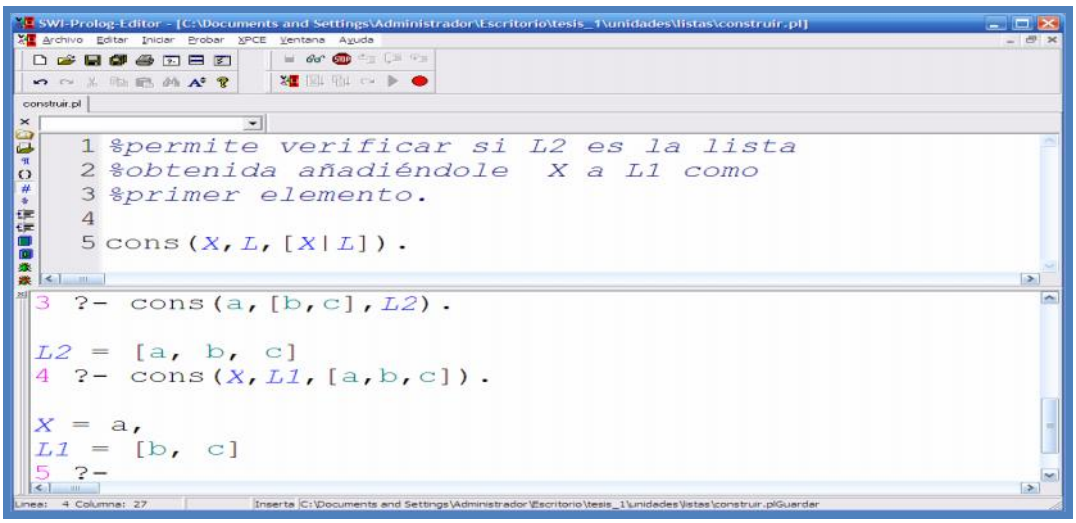


Figura 3. Construcción de listas.

3.3.2. Inserción de un elemento en una lista

Consiste en agregar un nuevo elemento a lista dada.

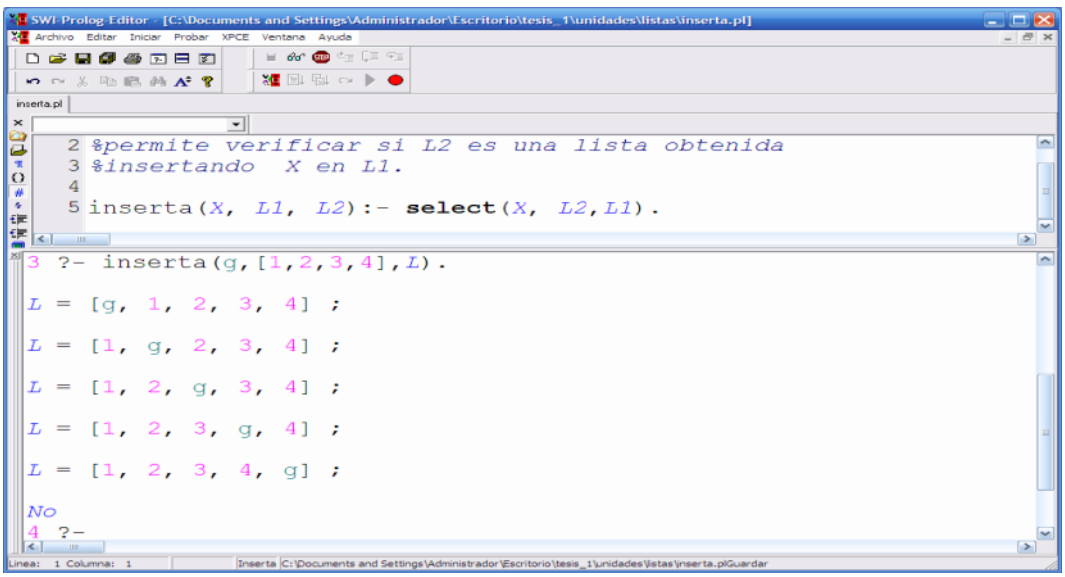


Figura 4. Inserción de un elemento en una lista.

3.3.3. Primer elemento

Para determinar si un elemento es el primero de una lista, este debe coincidir con la cabeza de la lista independientemente de lo que contenga el resto. Es por eso que ponemos una variable anónima en su lugar.

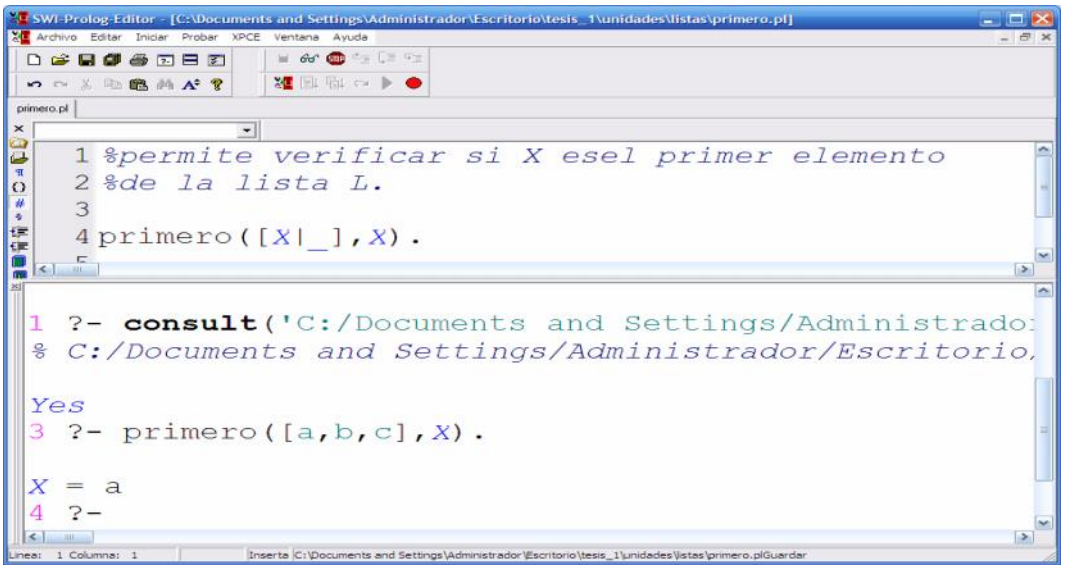


Figura 5. Primer elemento en una lista.

3.3.4. Último elemento

Para encontrar el último elemento de una lista dada, es un proceso que nos consume tiempo ya que uno tiene que recorrer la lista completa para encontrarlo.

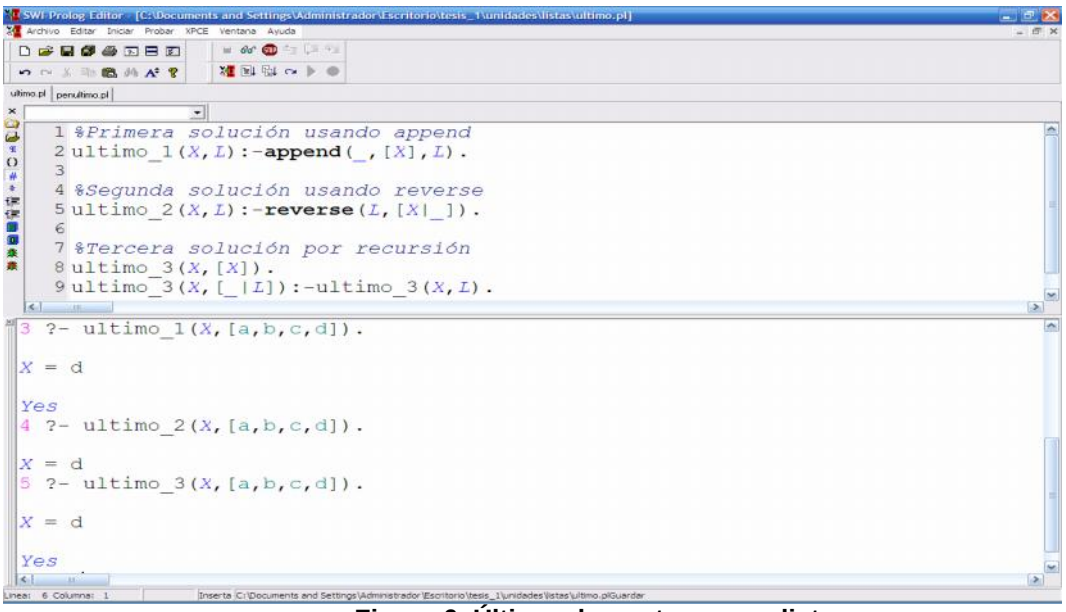


Figura 6. Último elemento en una lista.

3.3.5. Penúltimo elemento

El penúltimo elemento de una lista dada es el anterior elemento al último.

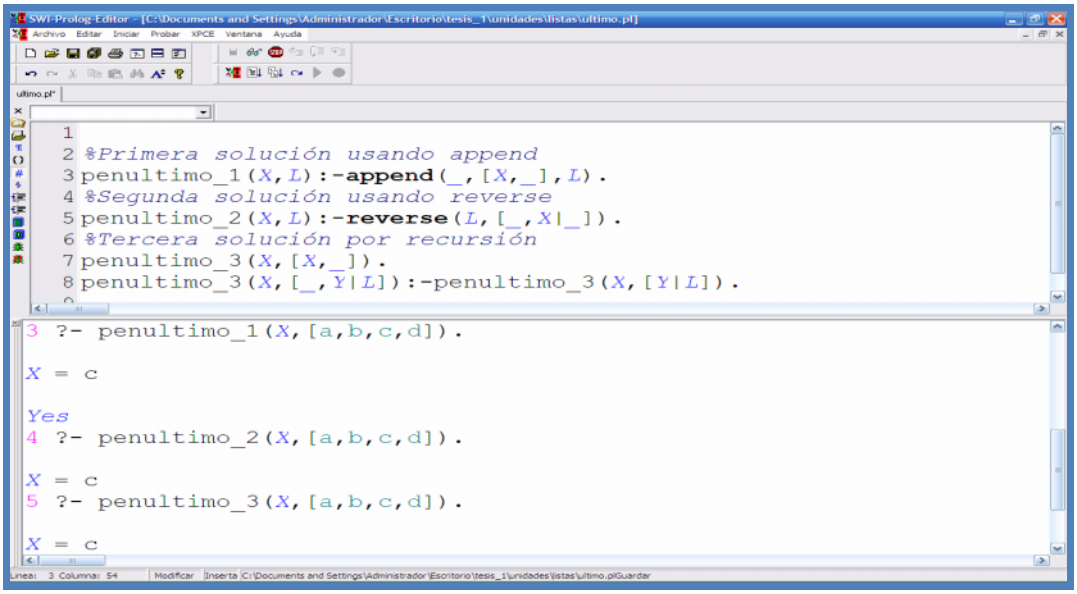


Figura 7. Penúltimo elemento en una lista.

3.3.6. Resto de una lista

Para determinar si un elemento es el resto de una lista este debe encontrarse en la cola de la lista.

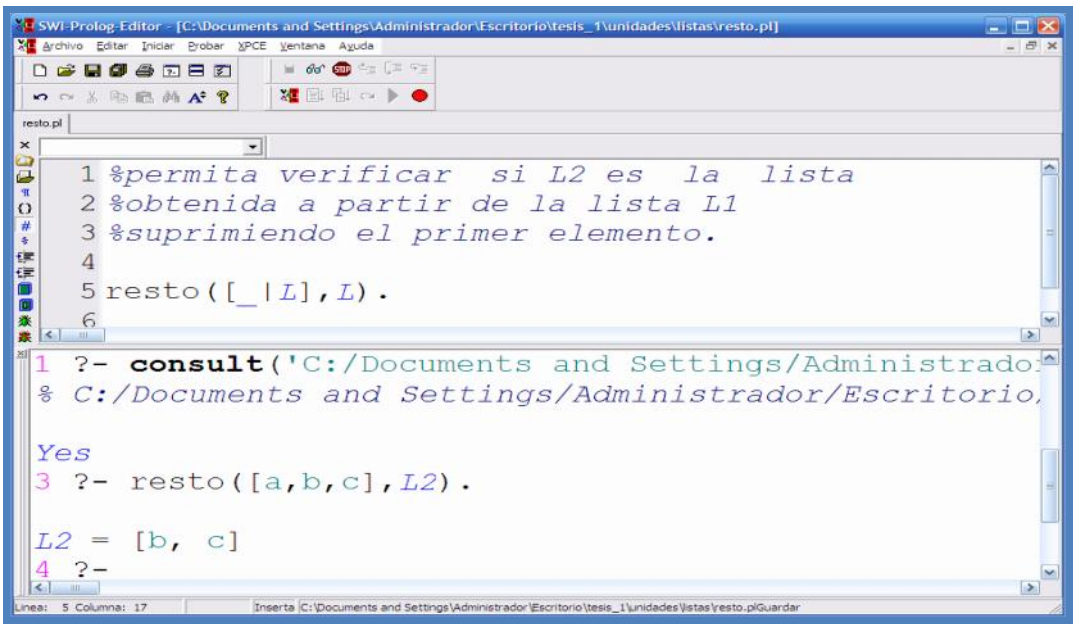


Figura 8. Resto de los elementos en una lista.

3.3.7. Relación de pertenencia

La relación de pertenencia es un claro ejemplo de la flexibilidad de las definiciones en Prolog ya que se pueden utilizar para:

1) Comprobar si un elemento pertenece a una lista.
2) Buscar un elemento de una lista.
3) Buscar una lista que contenga un cierto elemento.

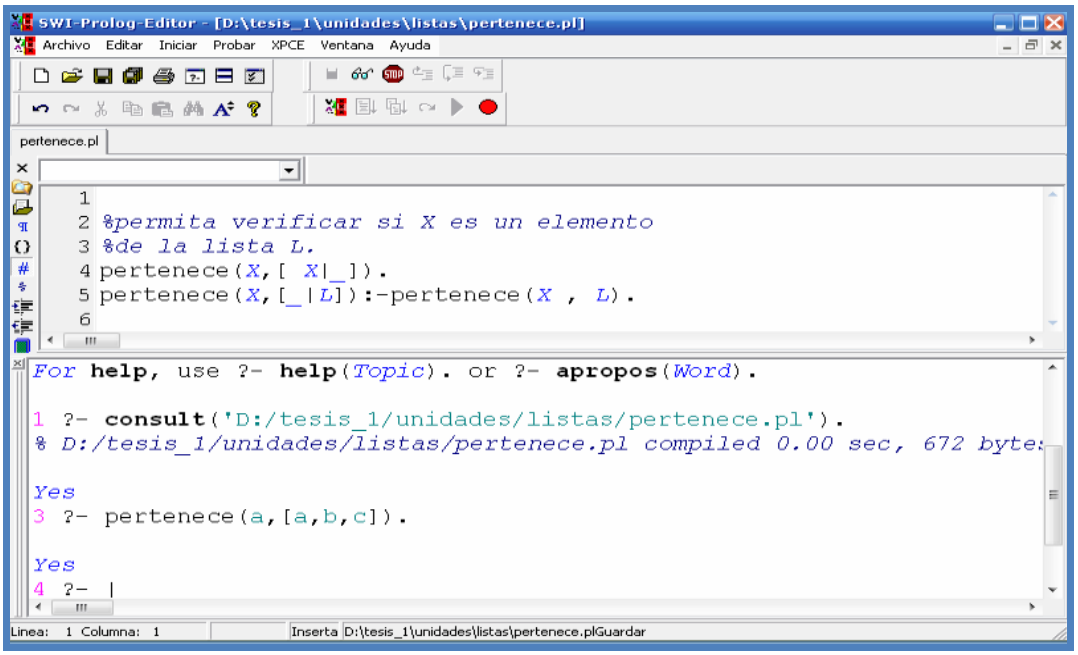


Figura 9. Relación de pertenencia de los elementos en una lista.

Para saber si un elemento pertenece a una lista:

1) El elemento debe de coincidir con la cabeza de la lista: Esto es independientemente de lo que contenga el resto de la lista, por lo que ponemos una variable anónima en su lugar. En este caso el predicado será cierto y Prolog no comprobara nada más.

2) El elemento debe de encontrarse en el resto de la lista: En la llamada recursiva de la segunda definición del predicado, la cabeza de la lista es una variable anónima, porque cuando se vaya a comprobar esta segunda regla, es seguro que la primera no se cumple, dado que el elemento no coincide con la cabeza de la lista y Prolog busca la cláusula en el orden de aparición en la base de datos.

3.3.8. Selección de un elemento

El predicado selecciona sirve para diversos propósitos:

1) Borrar elementos de una lista.

2) Insertar un elemento en diferentes posiciones de una lista.
3) Describir el comportamiento del predicado si se añade X\=Y en la segunda definición.

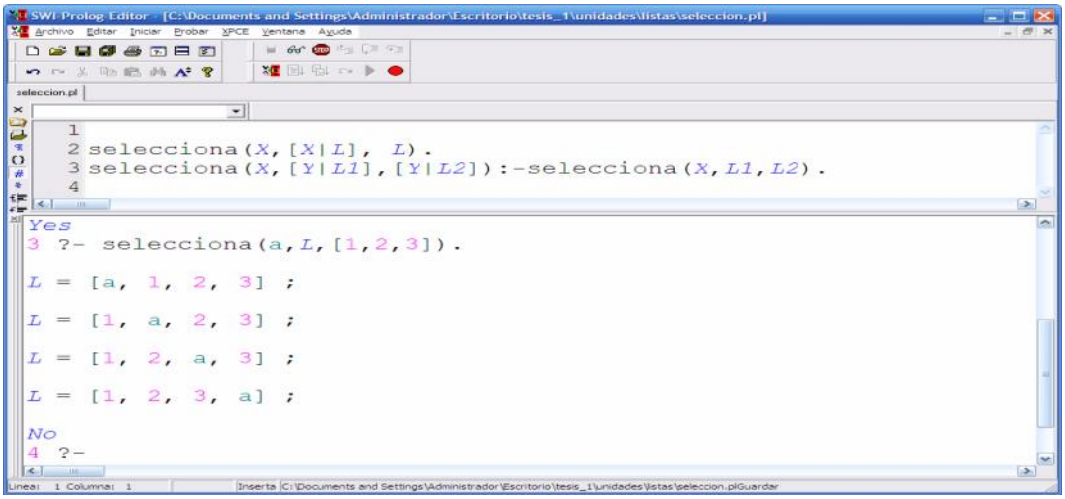


Figura 10. Selección de elementos de una lista.

3.3.9. Concatenación de listas

Concatenación es la unión de dos listas en el orden en que aparecen. Cuando concatenamos listas se pueden dar dos posibles casos:

1) Si el primer argumento es la lista vacía, entonces los argumentos segundo y tercero tienen que ser la misma lista.
2) Si el primer argumento es una lista no vacía, debe tener una cabecera y una cola, por ejemplo: [X|L], tal que procediendo de forma recursiva podremos concatenar las dos listas.

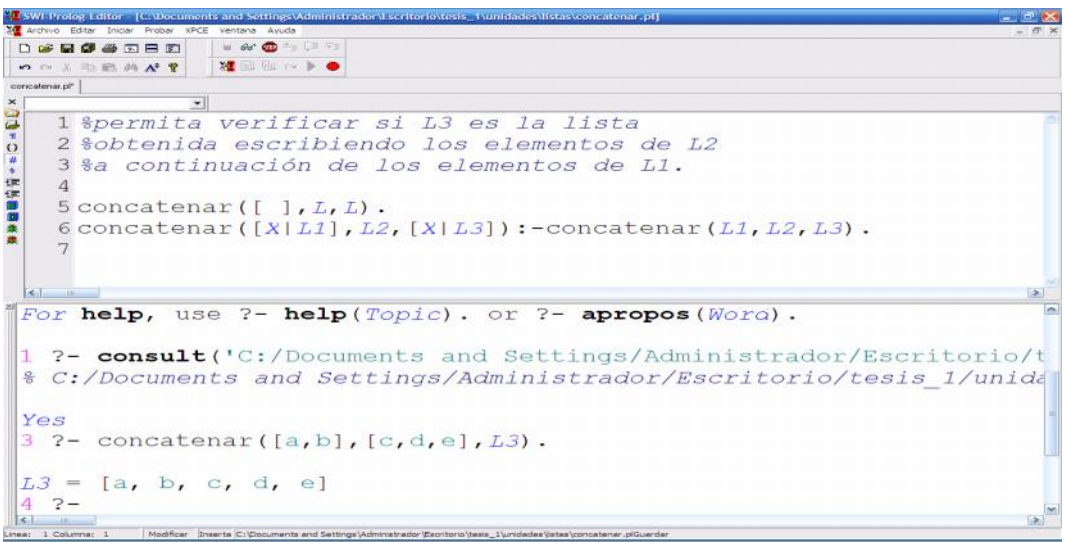


Figura 11. Concatenación de los elementos de una lista.

3.3.10. Lista inversa

Es la lista resultante al invertir los elementos de una lista. La cual haciendo uso de la relación append se puede calcular.

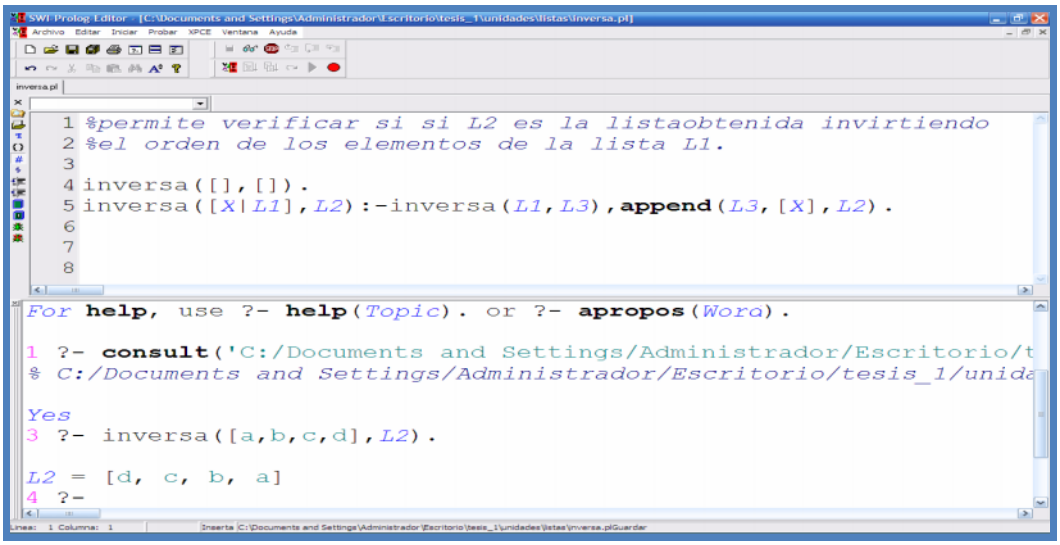


Figura 12. Lista inversa de los elementos de una lista.

Append: Es un predicado estándar predefino que recibe como argumento tres listas y es verdadero cuando la lista tres es el resultado de la concatenación entre la lista uno y la lista dos.

3.3.11. Palíndromo

Un palíndromo es una palabra o frase que se lee de igual forma de izquierda a derecha y de derecha a izquierda.

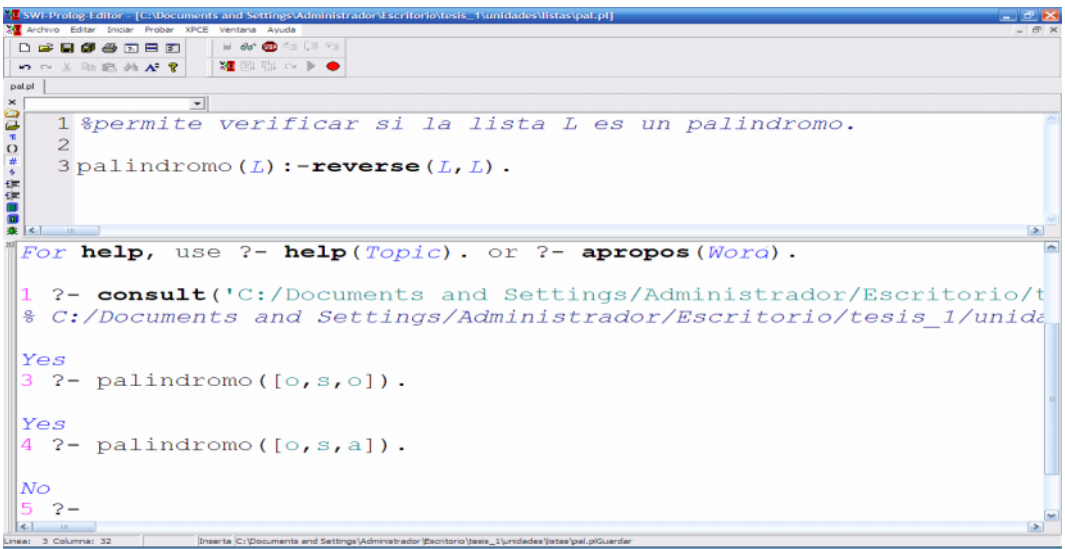


Figura 13. Palíndromo.

Reverse: Es un predicado estándar predefinido que es verdadero cuando la lista dos tiene los mismo elementos que la lista uno pero en un orden inverso a los elementos introducidos.

3.3.12. Permutación

La permutación es la acción y el efecto de permutar. Es decir cambiar los elementos de una lista por otra. La permutación de una lista permite obtener una nueva lista generada mediante permutaciones de los elementos de la otra.

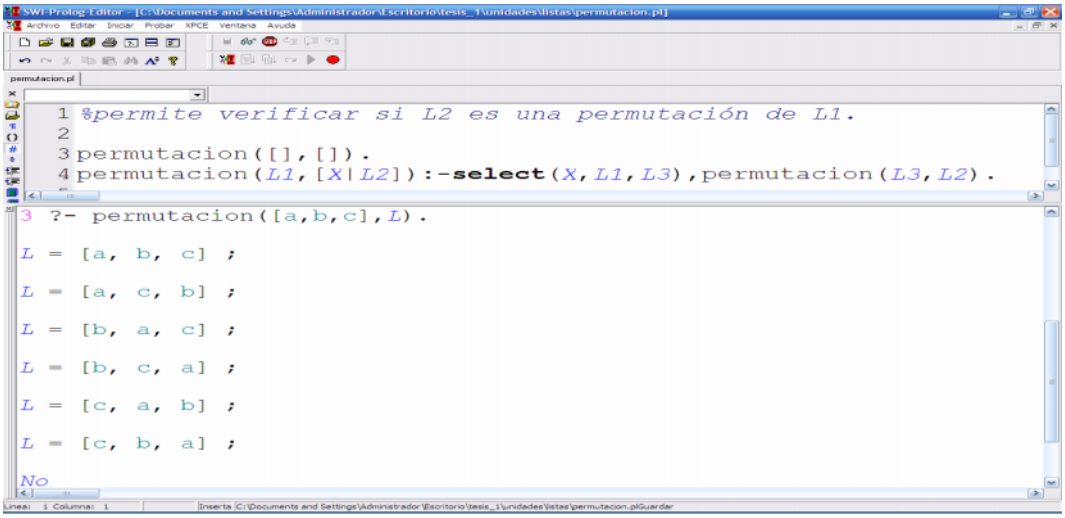


Figura 14. Permutación de los elementos de una lista.

3.3.13. Rotación de un elemento

Es cambiar la posición de los elementos seleccionados en una lista.

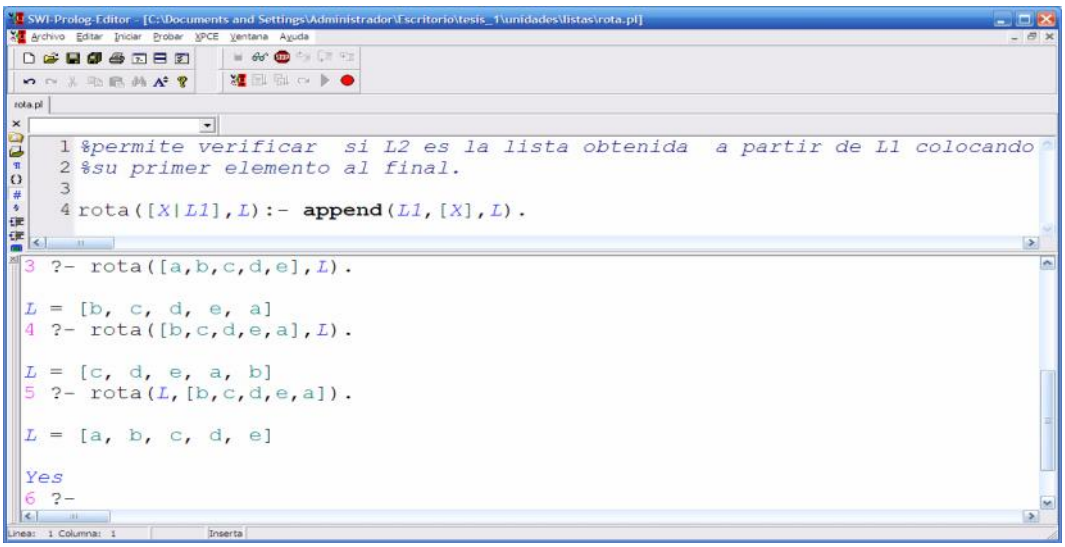


Figura 15. Rotación de un elemento.

3.3.14. Sublista

La relación de Sublista indica si una lista está contenida en otra. En el caso de este predicado nos indica que se satisface cuando el primer argumento es una sublista del segundo.

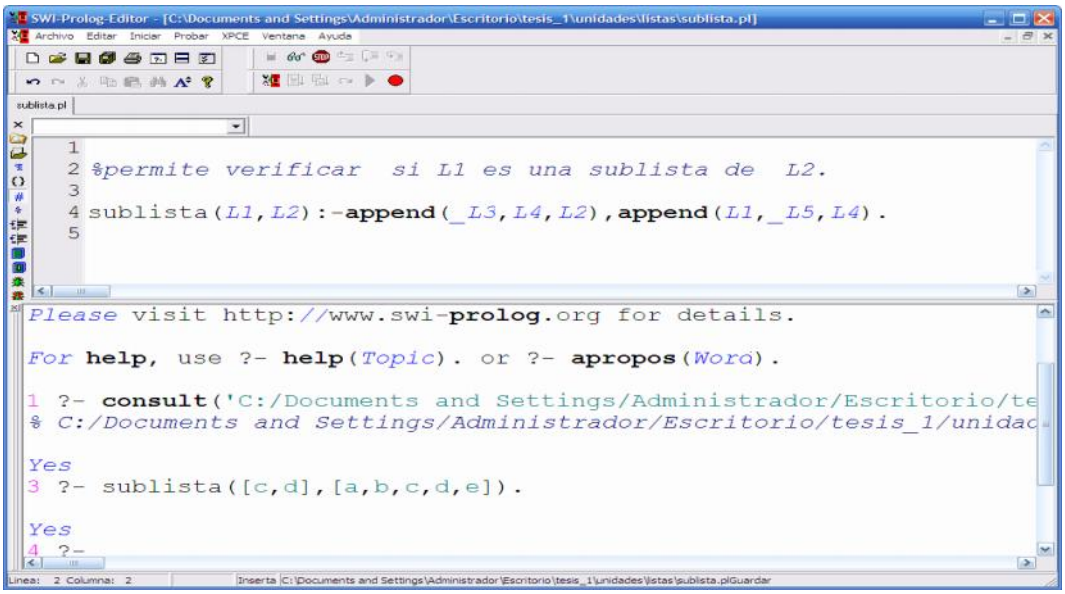


Figura 16. Sublista.

3.3.15. Subconjunto de una lista

Conjunto de elementos que pertenecen a otro conjunto.

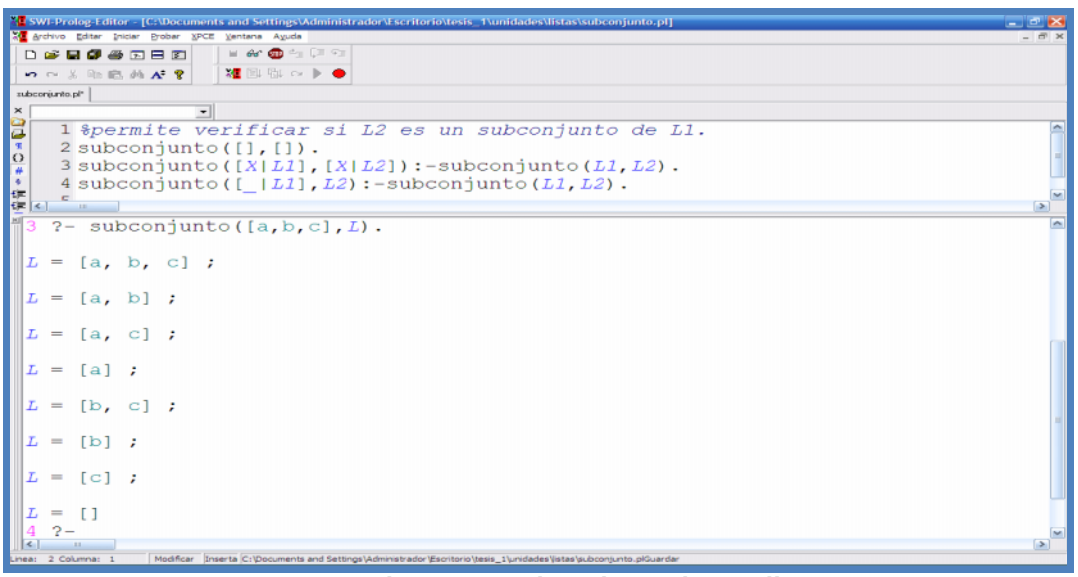


Figura 17. Subconjunto de una lista.

3.3.16. Lista con todos sus elementos iguales

Dada una lista cualquiera si todos sus elementos son similares se dice entonces que son iguales.

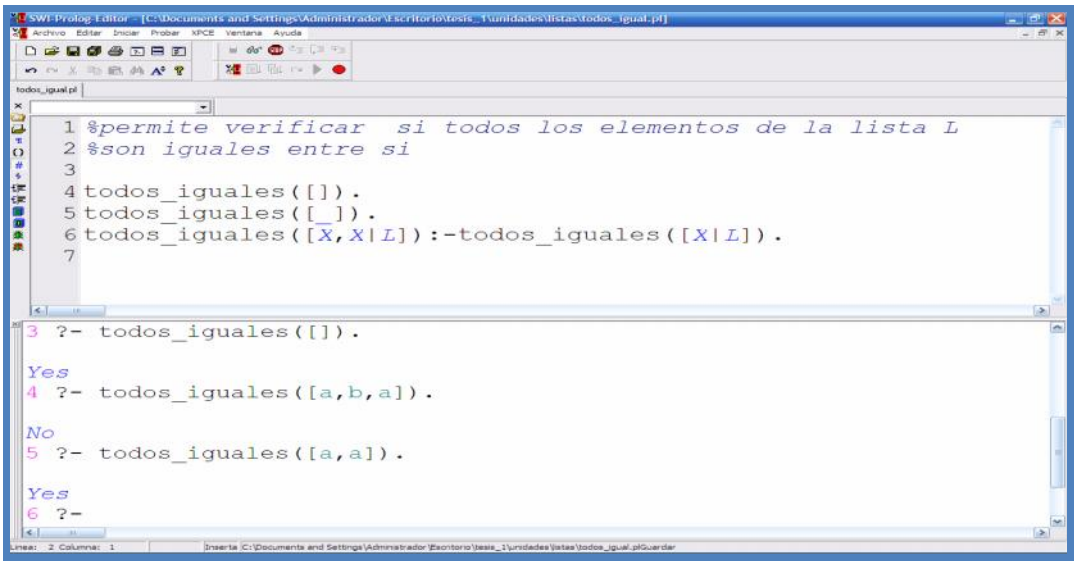


Figura 18. Lista con todos sus elementos iguales.

3.3.17. Paridad de la longitud de una lista

Permite comprobar si la cantidad de elementos contenidos en una lista son pares.

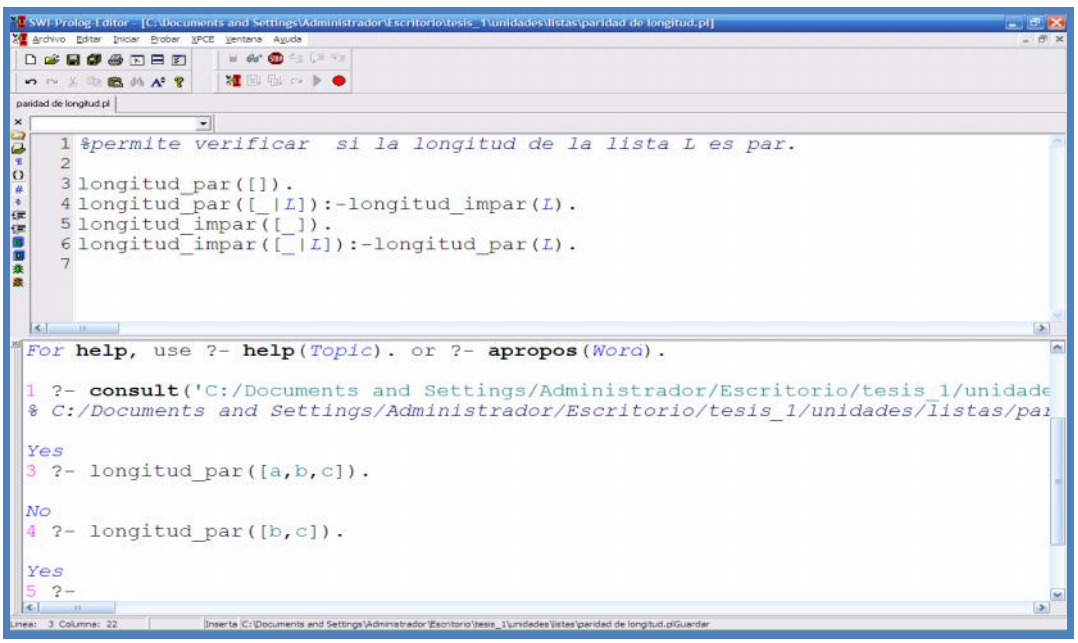


Figura 19. Paridad de la longitud de una lista.

3.4. Aritmética con lista

Las operaciones aritméticas pueden ser implementadas en una lista, en donde al introducir elementos de una lista, se puede obtener un valor a partir de ellas.

3.4.1. Suma

Es el resultado de añadir en una sola, varias cantidades de elementos pertenecientes a un mismo conjunto.

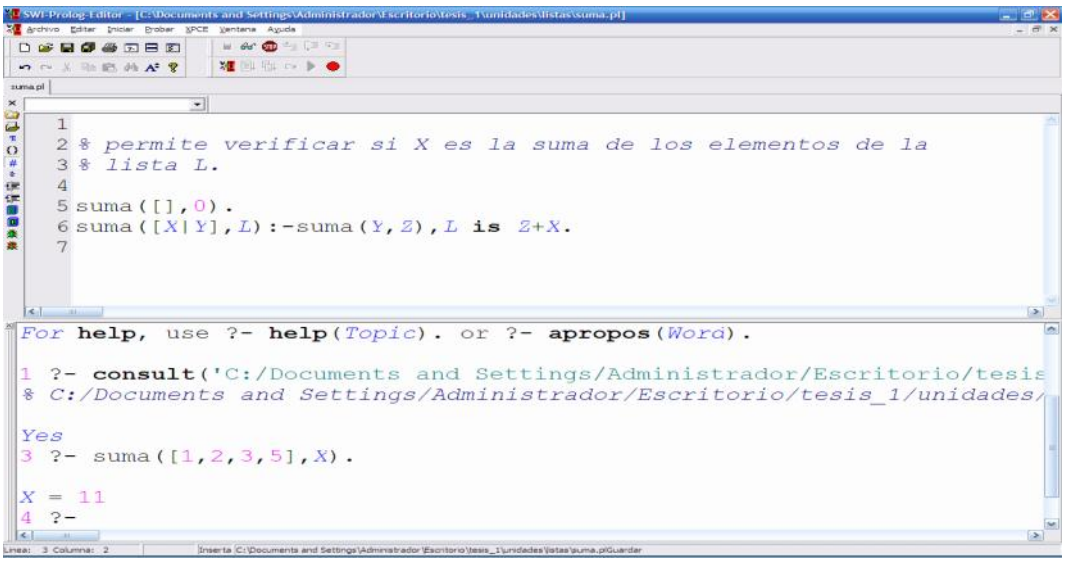


Figura 20. Suma de los elementos de una lista.

3.4.2. Producto

Es la cantidad que resulta al aumentar el número o la cantidad de elementos pertenecientes a un mismo conjunto.

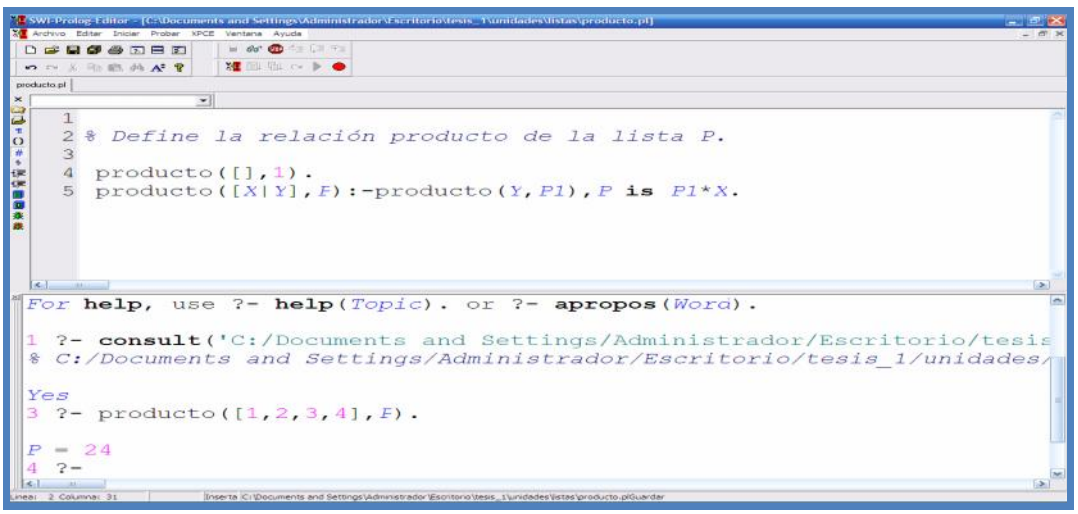


Figura 21. Producto de los elementos de una lista.

3.4.3. Longitud de lista.

Consiste en contar cada uno de los elementos que compone una lista.

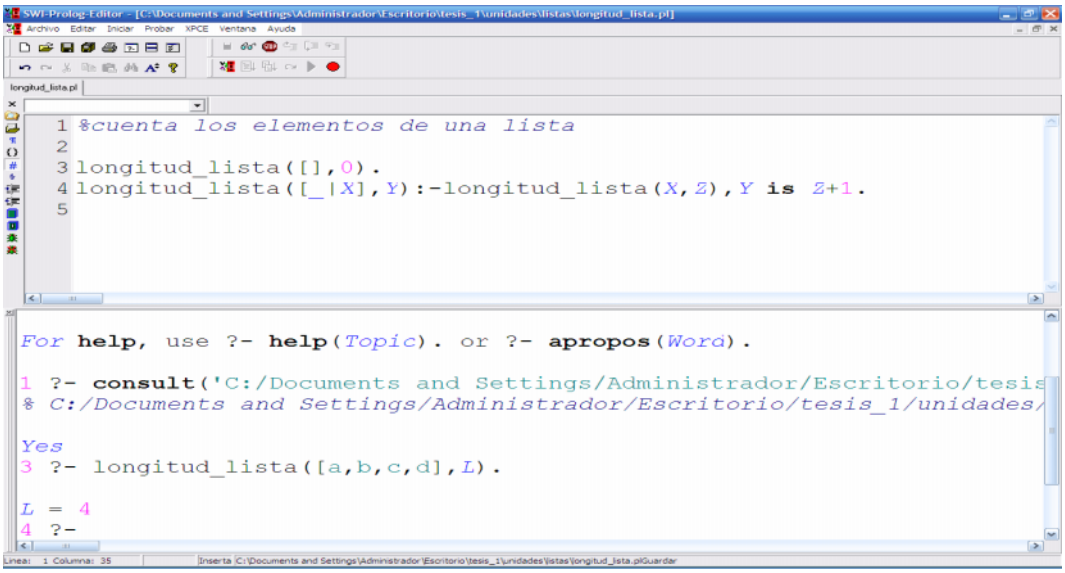
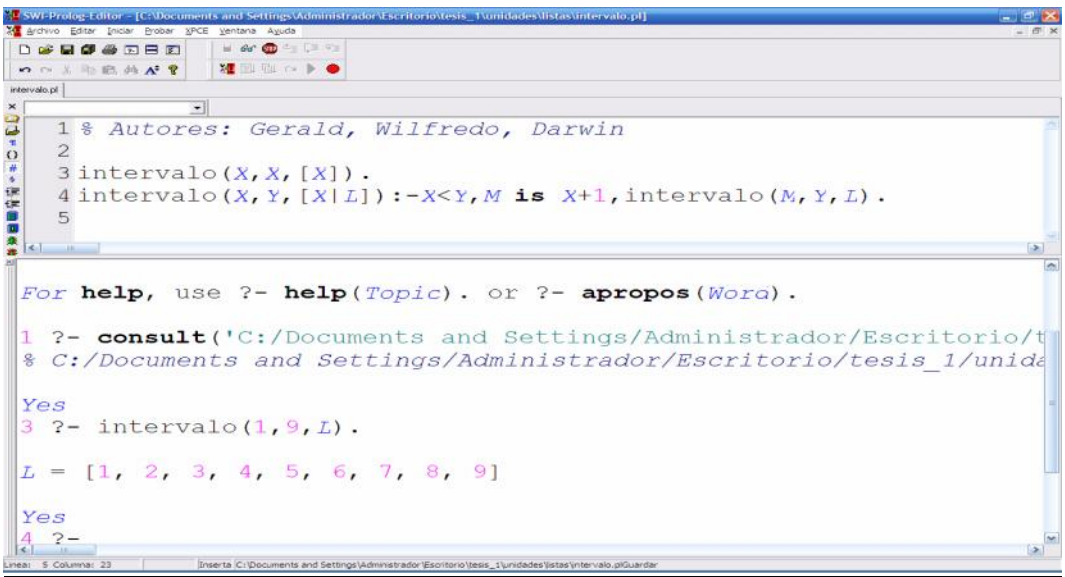


Figura 22. Longitud de los elementos de una lista.

3.5. Ejercicios resueltos

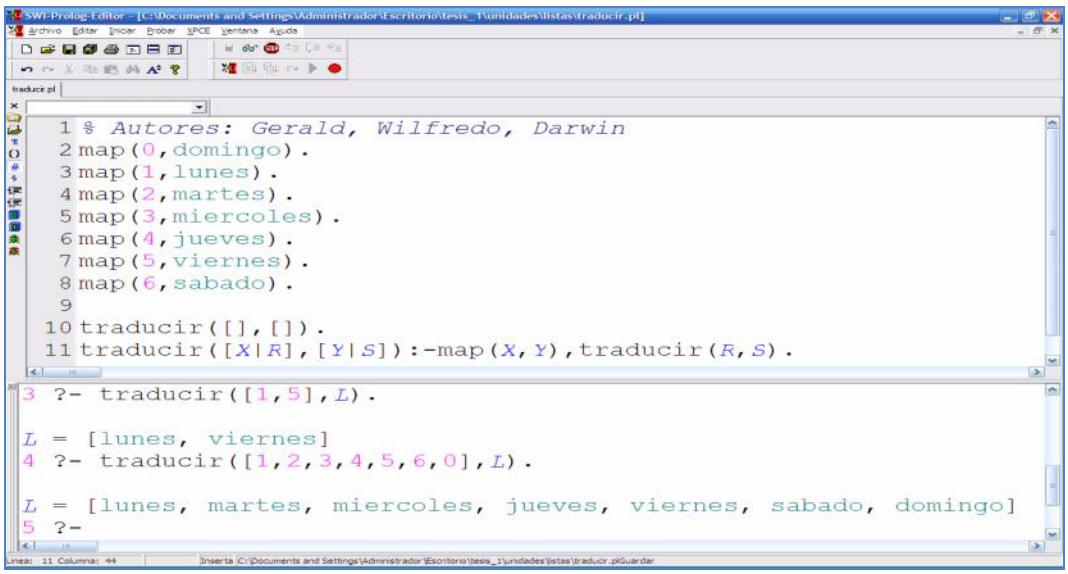
1) Defina la relación intervalo, si es posible generar una lista mediante la descomposición de un valor.

Solución:



2) Escriba una relación en Prolog que permita traducir una lista de números entre 0 y 6 inclusive en los correspondientes días de la semana, considerando el día domingo como 0 y el sábado como 6.

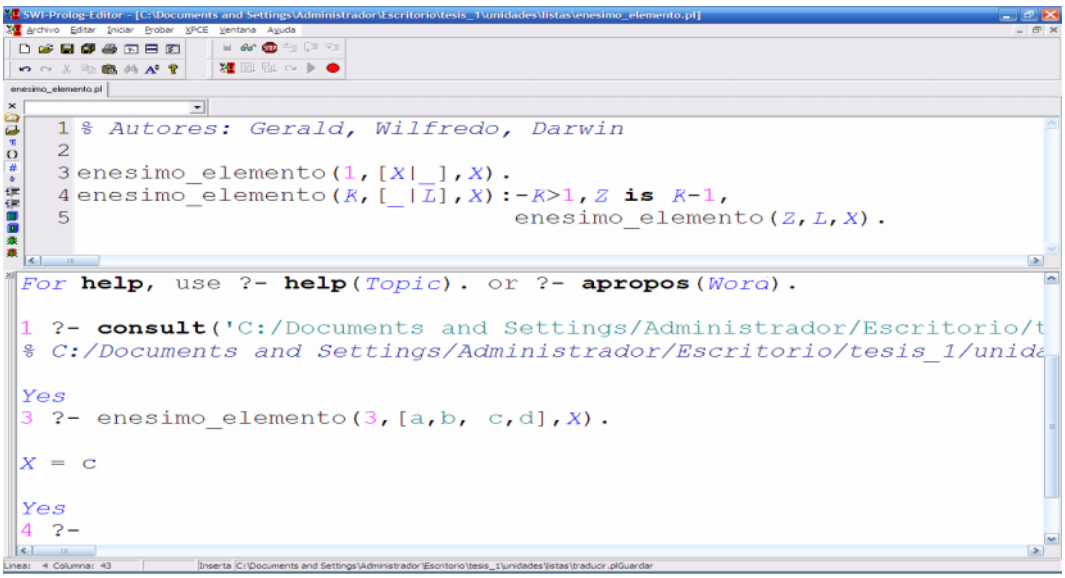
Solución:



3) Definir la relación enésimo _ elemento (K, L, X) que permita verificar si X es el K- ésimo elemento de la lista L.

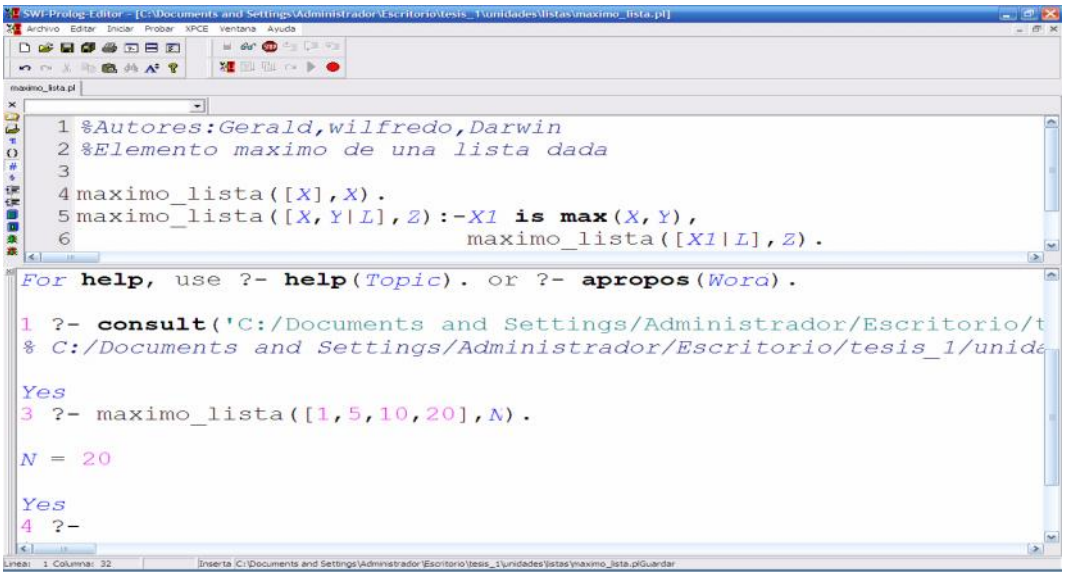
Nota: Recordar empezar a numerar en 1.

Solución:



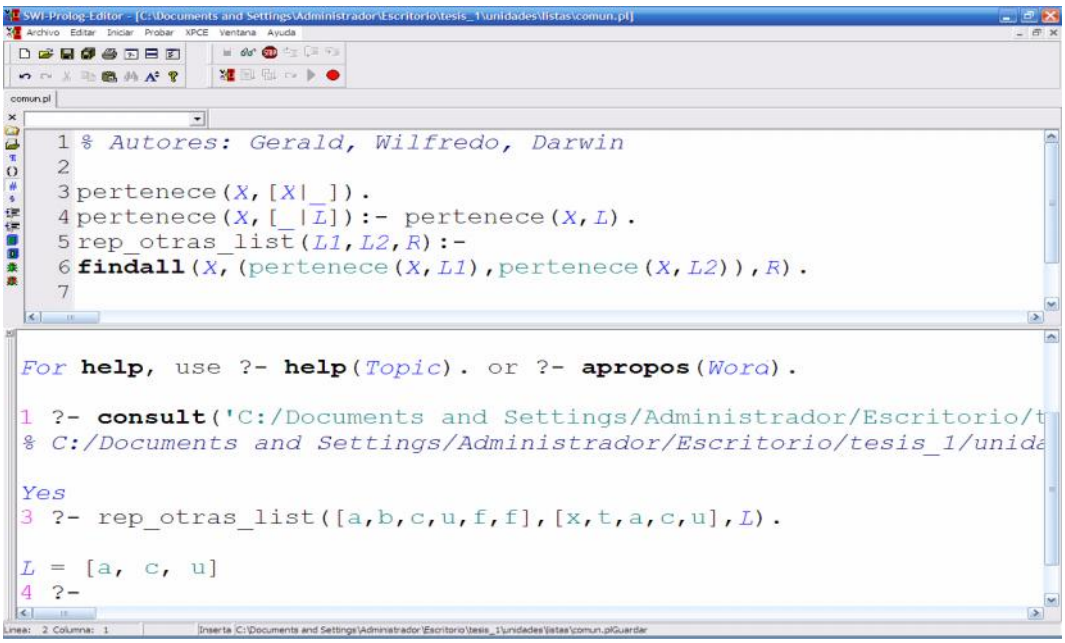
4) Definir la relación máximo _ lista (L, N) que se verifica si N es el máximo de los elementos de la lista de números L.

Solución:



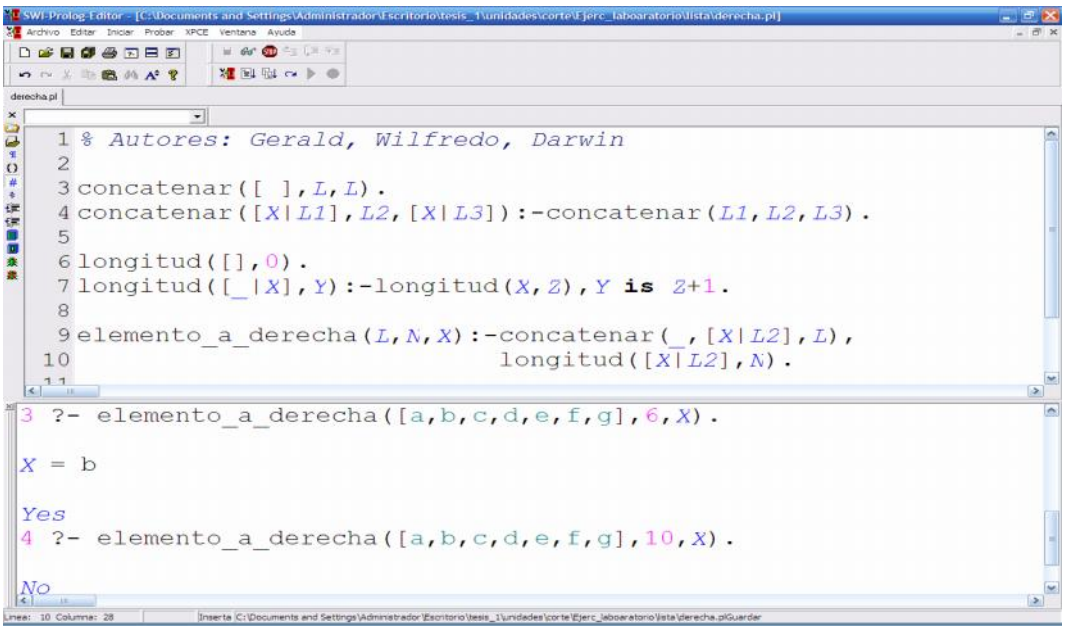
5) Escribe un predicado que permita obtener una lista compuesta por todos los elementos comunes a otras dos listas.

Solución:



6) Definir la relación elemento_a_ derecha (L, N, X) que permita verificar si X es el elemento situado en la N-ésimo posición, empezando a contar por la derecha, de la lista L.

Solución:



Tema 4: Estructuras de control

En este tema explicaremos las características del corte, fallo y la negación. Trataremos sus ventajas y desventajas y cuál es la utilidad que tienen como predicados predefinidos.

Contenido:

3. Estructuras de control
4.1. Reevaluación.
4.1.1. Introducción.
4.1.2. Definición de reevaluación.
4.1.3. característica de la reevaluación.
4.1.4. Conceptos relacionados a la reevaluación.
4.2. Predicado Corte (!).
4.2.1. Introducción
4.2.2. Definición Corte.
4.2.3. Representación del Corte.
4.2.4. Efecto del predicado Corte.
4.2.5. Utilidad del Corte.
4.2.6. Ventajas y desventajas del Corte.
4.3. Predicado Fallo (Fail).
4.3.1. Introducción
4.3.2. Definición.
4.3.3. Utilidad de Fail.
4.3.4. Ventajas y desventajas del uso de Fail.
4.4. Predicado Not (Negación).
4.4.1. Introducción.
4.4.2. Definición.
4.4.3. Utilidad de Not.
4.4.4. Ventajas y desventajas del uso de Not
4.5. Ejercicios resueltos.

4. Estructuras de control

4.1. Reevaluación

4.1.1. Introducción

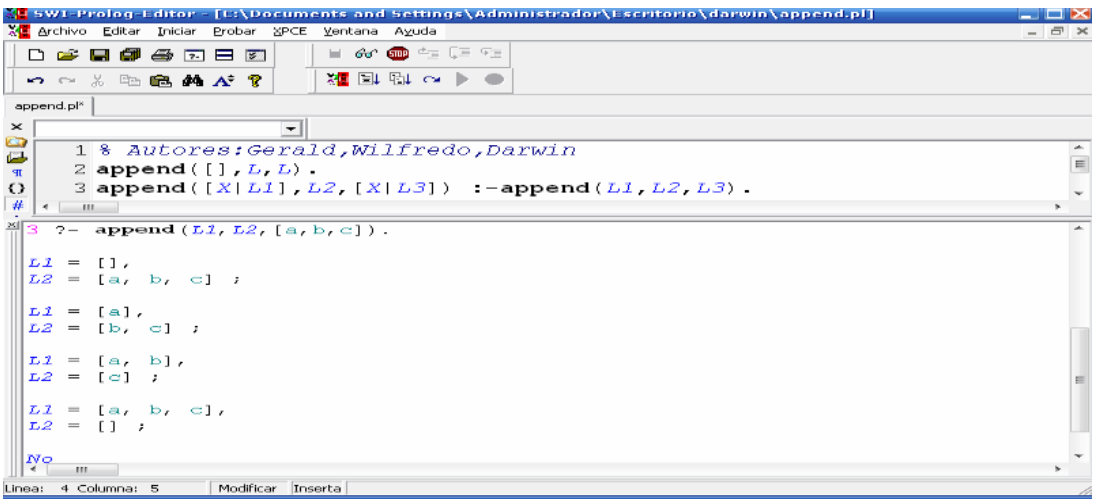
El mecanismo empleado por Prolog para satisfacer las cuestiones que se le plantean, es el de razonamiento hacia atrás (reevaluación o backtracking) complementado con la búsqueda en profundidad.

Prolog hace backtracking automáticamente si es necesario para volver a satisfacer un objetivo .El backtracking automático es un concepto de programación útil, ya que le facilita al programador la carga de programación con backtracking explicito.
Por otro lado, el backtracking incontrolado puede causar ineficiencia en un programa por lo que es preferible mejor controlarlo.

4.1.2. Definición de reevaluación

Es recordar el momento de la ejecución en que existen varias soluciones para dar marcha atrás y utilizar alguna otra solución como alternativa. Prolog de antemano conoce el numero de soluciones alternativas que puede tener, a cada solución se le denomina punto de elección, cada punto de elección se introducen en una pila de forma ordenada, se escoge el punto de elección y se ejecuta el proceso eliminando el punto de elección de la pila, si el objetivo tiene éxito se continua aplicando las mismas normas, si el objetivo falla Prolog da marcha atrás, recorriendo los objetivos que anteriormente tuvieron éxito y va descomponiendo la unión de sus variables. Cuando uno de los objetivos tiene un punto de elección anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa (elegir entre varias posibilidades la respuesta a la pregunta de un predicado).

Ejemplo:



4.1.3. Característica de la reevaluación

1) El mecanismo de vuelta atrás hace un recorrido en profundidad en un árbol.

2) El recorrido en profundidad regresa sobre sus pasos (retrocede) cada vez que encuentra un camino por el que no debe o no puede continuar.

3) Contiene dos dimensiones:

Altura: Es la cantidad de decisiones que debemos tomar.
Anchura: Es el número de alternativas que tenemos.

4.1.4. Conceptos relacionados con la reevaluación

1) Satisfacer un objetivo: Cuando Prolog intenta satisfacer un objetivo, busca en la Base de hechos desde su comienzo y:

- Si encuentra un hecho o la cabeza de una regla que pueda unificar con el objetivo, marca el lugar en la Base de hechos e instancia todas las variables previamente no instanciadas que coincidan. Si es una regla lo encontrado, intentará satisfacer los subobjetivos del cuerpo de dicha regla.

- Si no encuentra ningún hecho o cabeza de regla que unifique, el objetivo ha fallado, e intentaremos resatisfacer el objetivo anterior.

2) Resatisfacer un objetivo: Prolog intentará resatisfacer cada uno de los objetivos por orden inverso, para ello pretenderá encontrar una cláusula alternativa para el objetivo, en cuyo caso, se dejan sin instanciar todas las variables instanciadas al elegir la cláusula previa. La búsqueda la empezamos desde donde habíamos dejado el marcador de posición del objetivo.

4.2. Predicado Corte (!)

4.2.1. Introducción

El corte es un predicado predefinido que no tiene argumentos y cuya evaluación es siempre cierta. Se pueden incluir como un predicado más, en el cuerpo de las reglas o en las consultas.

Los cortes permiten al programador intervenir en el control del programa, puesto que su presencia hace que el sistema ignore ciertas ramas del árbol de resolución(Es una disposición en forma de árbol de los posibles caminos de resolución para una pregunta). Es un predicado útil para prevenir el backtracking. Y permite extender el poder de la expresividad de Prolog ya que admite la definición de un tipo de negación.

4.2.2. Definición Corte

El corte es uno de los predicados internos más polémicos del lenguaje Prolog. Se utiliza para “podar” ramas del árbol de resolución, consiguiendo que el sistema vaya más rápido. Ya que permite decirle a Prolog cuáles son las opciones previas que no hace falta que vuelva a considerar en un posible proceso de reevaluación.

El corte es un mecanismo muy delicado, que nos puede marcar la diferencia entre un programa que funcione y otro que no. Un mal uso de este predicado es que puede podar ramas del árbol de resolución que contengan soluciones, impidiendo que el sistema encuentre algunas soluciones o todas a un problema dado. De esta forma se aconseja utilizar el corte con precaución y únicamente cuando se necesite.

4.2.3. Representación del Corte

El predicado corte se representa mediante el símbolo ”!” que se satisface inmediatamente y no puede volver hacerlo. Es decir, se convierte en una valla que impide que la reevaluación la atraviese en su vuelta hacia atrás, convirtiendo en inaccesibles todos los marcadores de las metas que se encuentran a su izquierda.

4.2.4. Efecto del predicado Corte

1) El predicado siempre se cumple.
2) Si se intenta volver a ejecutar(al hacer backtracking) elimina las alternativas restantes de los objetivos que hay desde su posición hasta la cabeza de la regla donde aparece.

4.2.5. Utilidad del predicado Corte

La utilidad del predicado corte viene dada por:

- Optimización del tiempo de ejecución.
- Optimización de memoria.
- Confirmación de la elección de una regla.
- Advertencia de que se va por mal camino.
- Terminación de la generación de soluciones múltiples.

4.2.6. Ventajas y desventajas del predicado Corte

Ventajas:

- Permite aumentar la expresividad del lenguaje.

- Permite mejorar la eficiencia de los programas, evitando la exploración de partes del árbol de resolución de las que se sabe de antemano que no conducirán a ninguna nueva solución.

- Permite implementar algoritmos diferentes según la combinación de argumentos de entradas. Algo similar al comportamiento de las sentencias case en los lenguajes imperativos.

Desventajas:

- Al tratarse de una herramienta de control, interviene en cómo se debe de resolver el problema, por lo que su uso entra en clara contradicción con los principios básicos de la programación lógica pura.

- Puede conducir a programas lógicos difíciles de leer y validar.

- Puede provocar muchos errores de programación.

4.3. Predicado Fallo (fail)

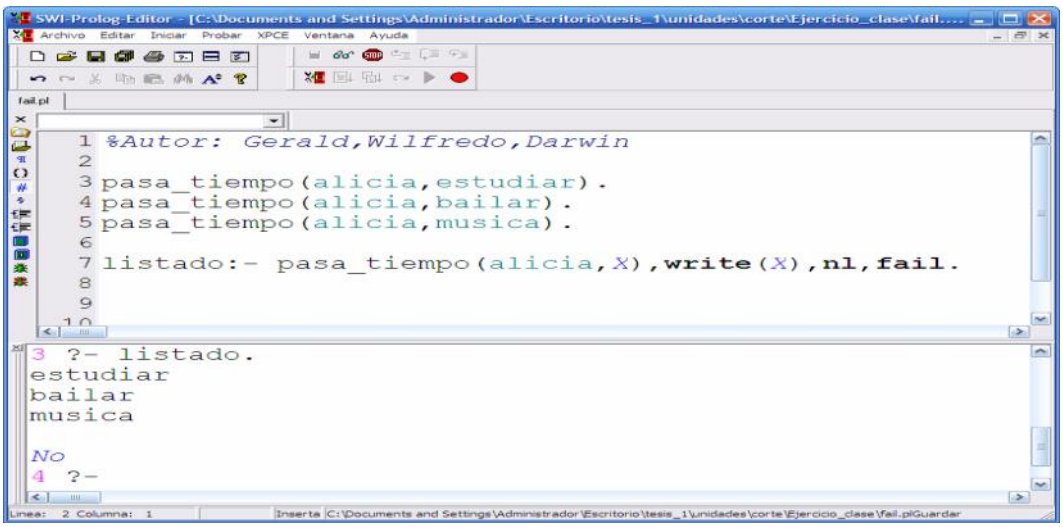
4.3.1. Introducción

Fail es un predicado que siempre falla, por tanto, implica la realización del proceso de retroceso para que se generen nuevas soluciones. Cuando la máquina Prolog encuentra una solución para y devuelve el resultado de la ejecución. Con Fail podemos forzar a que no pare y siga construyendo el árbol de búsqueda hasta que no queden más soluciones que mostrar.

4.3.2. Definición

Fail es un predicado predefinido que no tiene argumentos y permite la generación de todas las posibles soluciones para un problema dado.

A continuación se presenta un ejemplo en donde se puede ver el funcionamiento de este predicado.



Proceso de ejecución

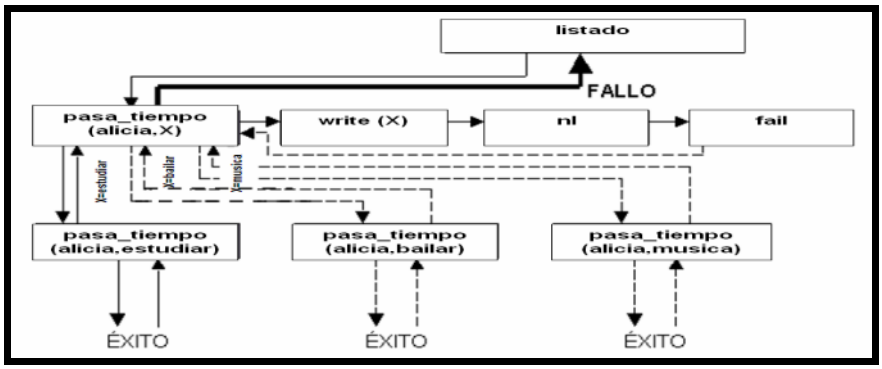


Figura 23. Ilustración del funcionamiento del predicado Fail

4.3.3. Utilidad del predicado Fail

- Útil para obligar a Prolog a dar un fallo.
- Útil para forzar la vuelta atrás (backtracking).
- Se utiliza en combinación con el corte para el tratamiento de excepciones.
- Se utiliza para detectar prematuramente combinaciones de los argumentos que no llevan a solución, evitando la ejecución de un montón de código que al final va a fallar de todas formas.

4.3.4. Ventajas y desventajas de Fail

Ventajas:

- Permite generar todas las posibles soluciones a un problema dado.
- Evita la aplicación de una regla, ya que en combinación con el Corte puede forzar el fallo.
- Permite detectar casos explícitos que invalidan un predicado.

Desventajas:

- Deficiencia en tiempo de ejecución.
- Deficiencia de memoria.
- Puede producir resultados no deseados.

4.4. Predicado Negación (not)

4.4.1. Introducción

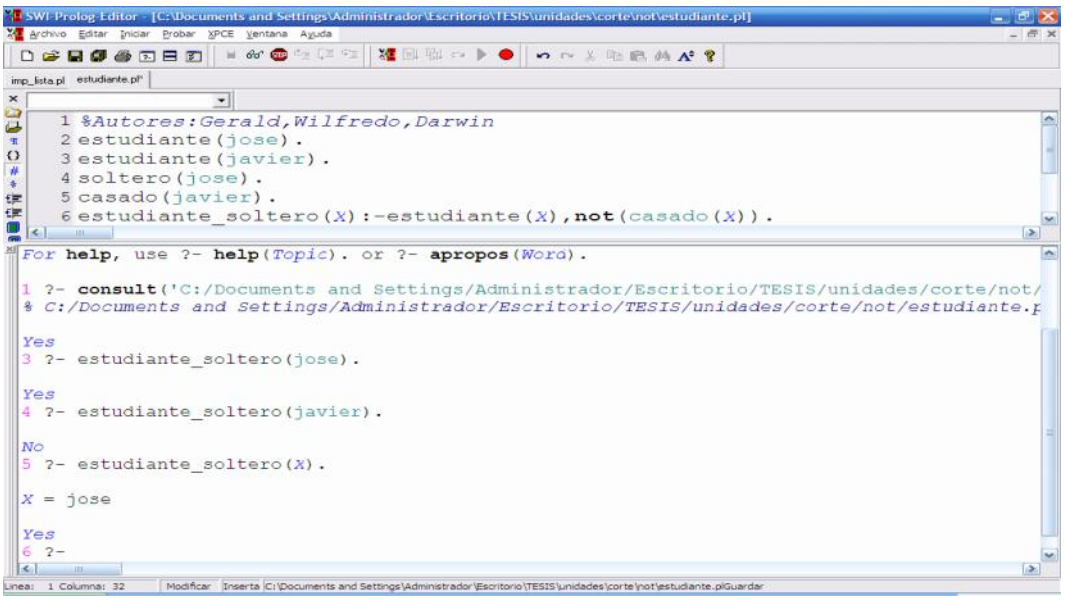
El lenguaje Prolog utiliza un subconjunto de la lógica de predicados de primer orden (cláusulas Horn) lo que impide modelar ciertas situaciones con conocimiento negativo. La negación por fallo es una aproximación a las técnicas de representación de conocimiento negativo.

4.4.2. Definición

La negación en el lenguaje Prolog se realiza mediante el predicado **not** el cual está implementado como negación por fallo. Este predicado antes de la llamada a un predicado P cambia su valor de verdad, es decir, si el predicado P tiene éxito, not (P) fallará y si el predicado P falla, not (P) tendrá éxito.

Cuando se haga uso de not se debe tener la precaución de aplicarlo únicamente en llamadas donde todas las variables existentes estén ya instanciadas, ya que si no el comportamiento no será el esperado.

Ejemplo:



4.4.3. Implementación de not

Una de las formas más extendidas de implementar la negación en lógica es mediante la utilización de call (El cual fuerza la comprobación de P como si se tratara de una consulta realizada al intérprete), el corte y el predicado predefinido fail. En este contexto, la utilización del corte y de dicho predicado, permiten la definición de un predicado not (P) que es cierto cuando P no se puede demostrar y es falso en otro caso.

```
not (P):-call (P),!, fail.
not (P).
```

Para comprender la implementación de la negación nos vamos a fijar en la Figura 4.2 que muestra el funcionamiento del predicado cuando P es cierto y en la Figura 4.3 que muestra el funcionamiento cuando P es falso.

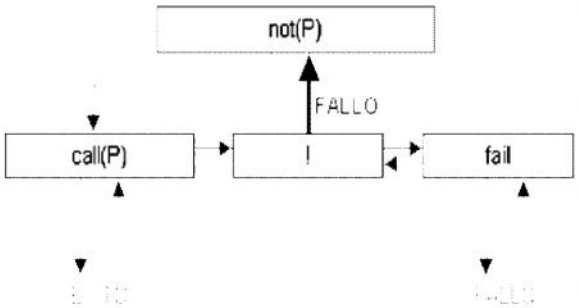


Figura 24. Funcionamiento de not cuando p es cierto

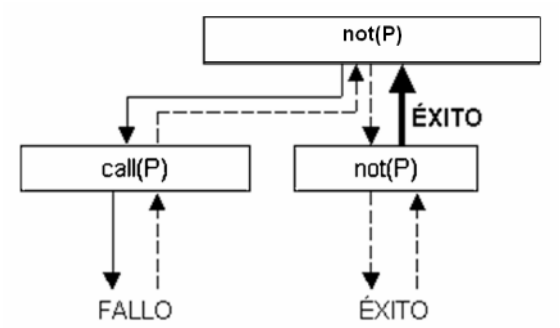


Figura 25. Funcionamiento de not cuando p es falso

En donde la terminación de not (P) depende de la terminación de P. Ya que se puede ver en la figura 4.2 que cuando P tiene éxito, con fail se provoca el fallo y antes de esto se cierran las alternativas con un corte(es decir not (P) falla sin posibilidad de utilizar otra cláusula). Por el contrario si en la ejecución de P aparece al menos una solución a la izquierda de la primera rama infinita, not (P) terminará con éxito (figura 4.3).

4.4.4. Ventajas y desventajas de la negación

Ventajas:

- Permite aumentar la eficiencia de los programas.
- Permite expresar reglas que son mutuamente excluyentes.

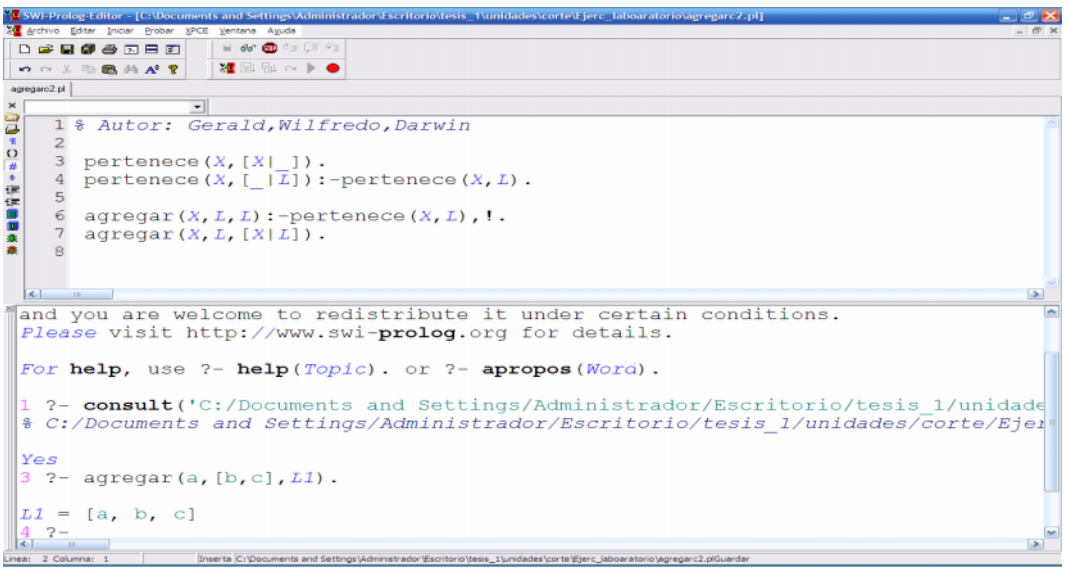
Desventajas:

- Se pierde correspondencia entre significado declarativo y procedural.
- El cambio del orden de las cláusulas puede afectar el significado declarativo.
- No corresponde a una negación lógica, sino al hecho de que no hay evidencia para demostrar lo contrario.

4.5. Ejercicios resueltos

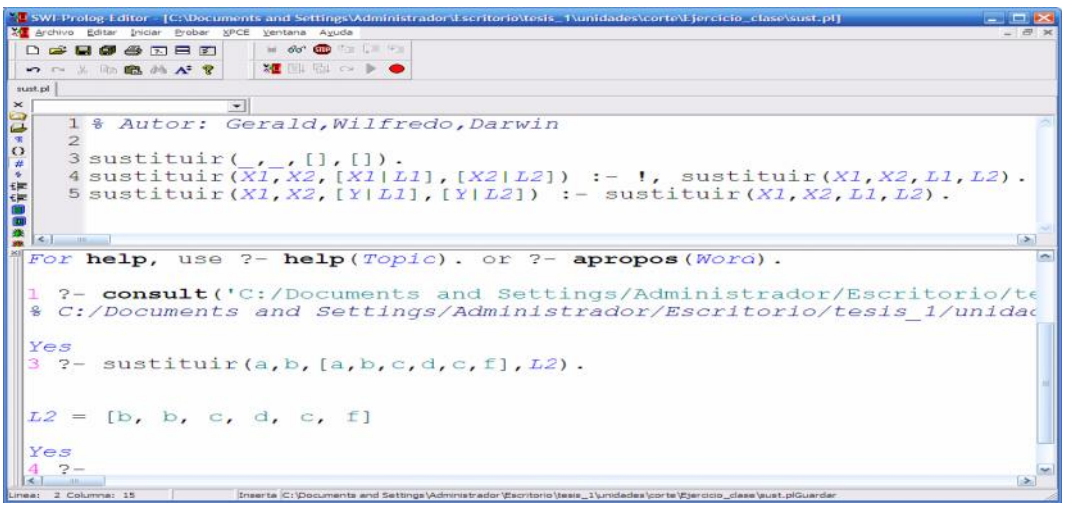
1) Definir la relación agregar (X, L, L1) que permita verificar si L1 es la lista obtenida añadiendo X a L, si X no pertenece a L y es L en caso contrario.

Solución:



2) Definir la relación sustituir (X, Y, L1, L2) que permita verificar si L2 es la lista del resultado de sustituir en la lista L1 todas las ocurrencias del elemento X por el elemento Y.

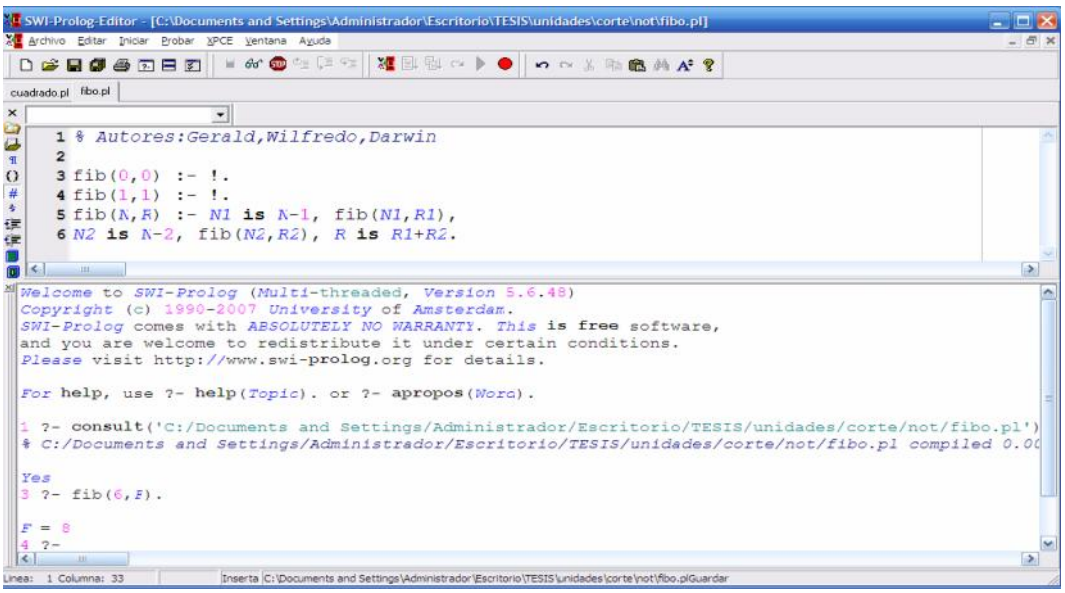
Solución:



3) Definir la relación fibonacci(N, F) que permita verificar si F es el N–ésimo término de la sucesión Fibonacci.

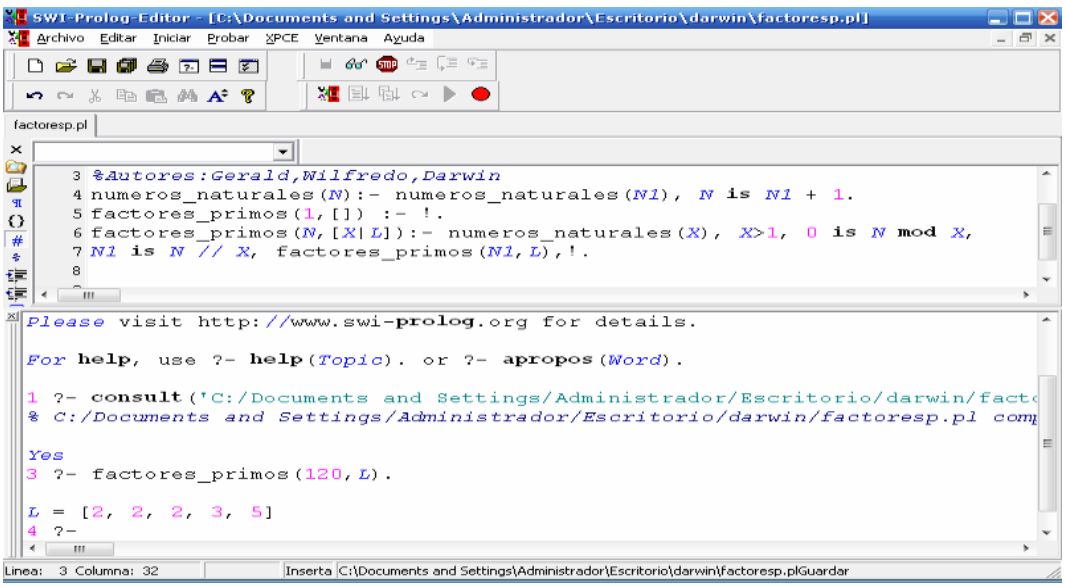
Nota: La sucesión Fibonacci es 0, 1, 1, 2, 3, 5, 8, 13, 21,. . . en la que cada término, salvo los dos primeros, es la suma de los dos anteriores.

Solución:



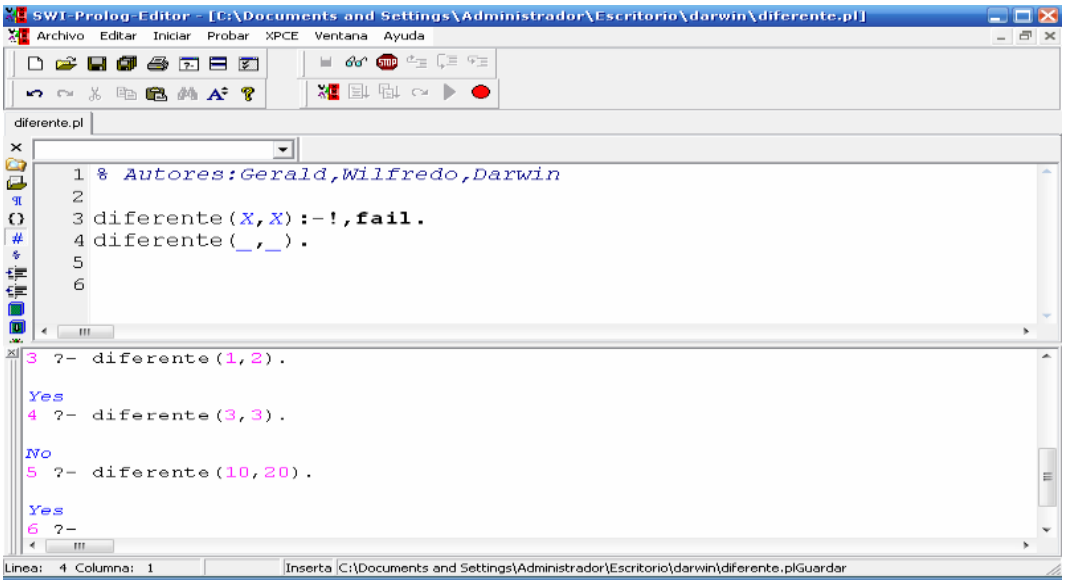
4) Definir la relación factores_primos(N, L) que permita verificar si L es la descomposición en factores primos del numero N.

Solución:



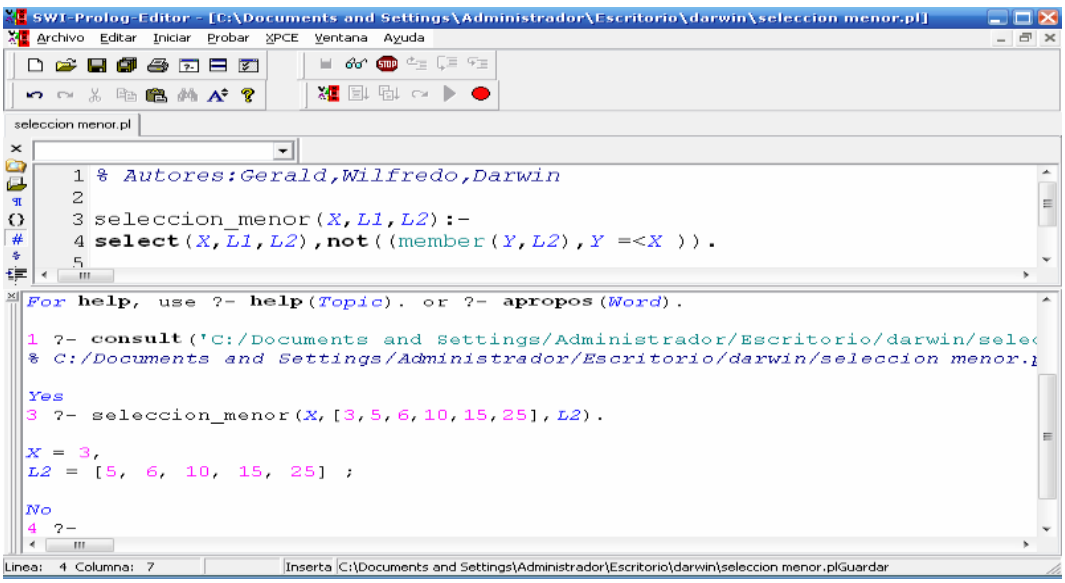
5) Definir el predicado diferente(X, Y) que sea cierto cuando sus dos argumentos no sean unificables.

Solución:



6) Definir la relación selección _ menor(X, L1, L2) que permita verificar si X es el menor elemento de la lista L1 y L2 contiene al resto de los elementos.

Solución:

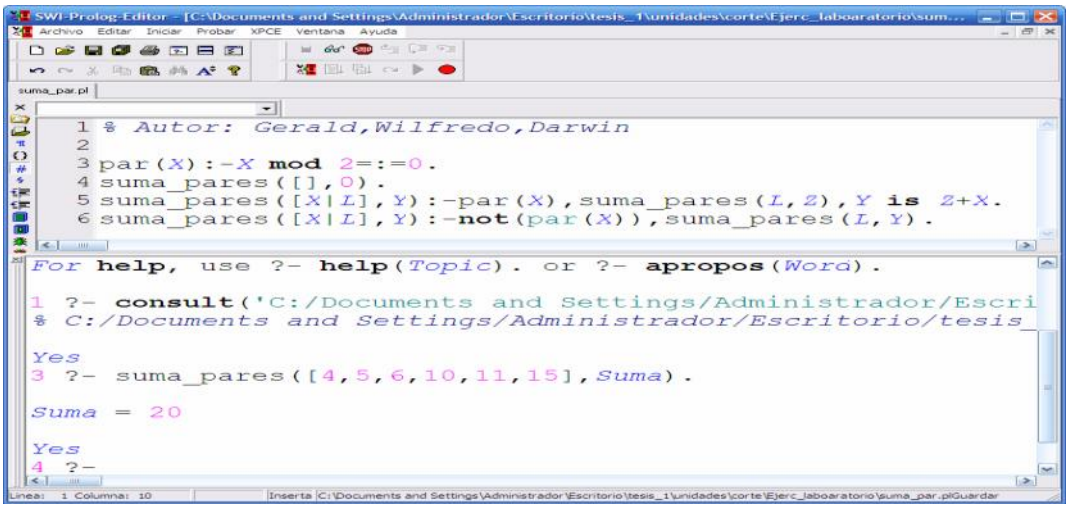


7) Definir la relación suma _ pares (L, S) que permita verificar si S es la suma de todos los números pares de la lista de números L.

Nota:

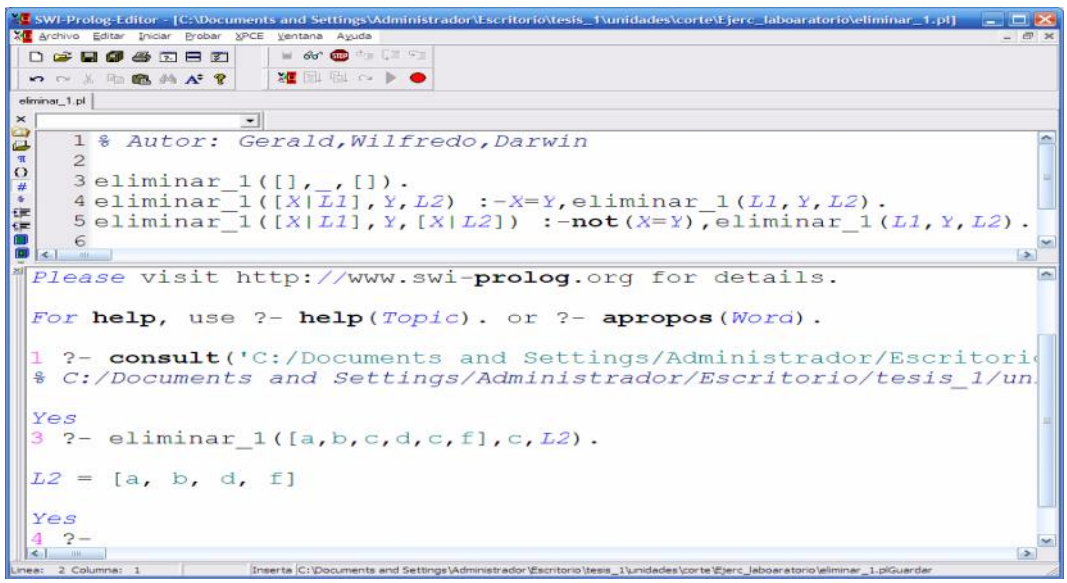
Recordar definir cuando un número es par, para la resolución del ejercicio.

Solución:



8) Definir la relación eliminar (L1, X, L2) que permita verificar si L2 es la lista obtenida eliminando los elementos de L1 unificables simultáneamente con X.

Solución:



Tema 5: Árboles

En este tema se abordará el concepto de árboles, el cual es muy importante ya que estos son usados en diferentes aplicaciones. Se aprenderá el manejo y las operaciones con árboles, utilizando conceptos estudiados en unidades anteriores como listas, recursividad, etc.

Contenido:

5. Árboles
5.1. Definición de árboles.
5.2. Representación de árboles.
5.3. Ventajas y desventajas del uso de árboles.
5.4. Recorrido de árboles.
5.5. Características del recorrido de árboles
5.6. Árboles binarios
5.6.1. Árboles binarios de búsqueda.
5.6.2. Operaciones sobre árboles binarios de búsqueda
5.7. Ejercicios resueltos.

5. Árboles

5.1. Definición de árboles

Un árbol es una estructura de datos formada por un conjunto de nodos y un conjunto de ramas representado mediante un grafo acíclico (no tiene ciclos), conexo (existe exactamente un camino entre todo par de nodos) y no dirigido. En un árbol existe un nodo especial denominado raíz. Un nodo del que sale alguna rama recibe el nombre de nodo de bifurcación o nodo rama y un nodo que no tiene ramas recibe el nombre de nodo terminal o nodo hoja.

5.2. Representación de árboles

Utilizando la siguiente regla:

arbol(libro,arbol(titulo,arbol(prolog,vacio,vacio),vacio),
arbol(autores,arbol(gerald,vacio,vacio),arbol(wilfredo,vacio,vacio))).

Los dos tipos de representaciones serían las siguientes:

1. Mediante un grafo:

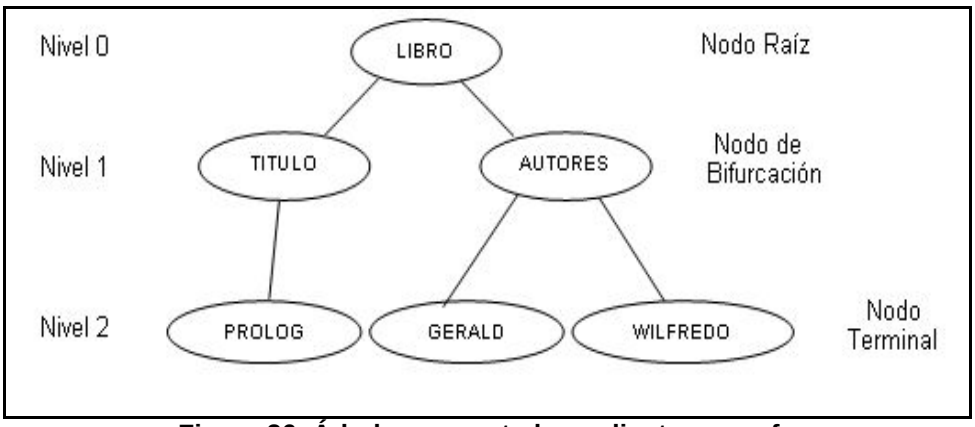


Figura 26. Árbol representado mediante un grafo

2. Mediante un diagrama encolumnado:

Figura 27. Árbol representado mediante un diagrama encolumnado.

5.3. Ventajas y desventajas del uso de árboles

Ventajas:

- Establece de ante mano las políticas a seguir, dada la ocurrencia de ciertos eventos.
- Su funcionamiento es igual al de la programación lógica o declarativa, en la cual se establecen hechos para obtener el resultado óptimo.
- La forma de un árbol es intuitiva, esto significa que el usuario puede comprobar la racionalidad del modelo y si lo cree necesario, modificar el árbol o influir en su arquitectura.

Desventaja:

- El tamaño del árbol se incrementa rápidamente en cuanto el número de nodos crece, al ocurrir esto se presentan muchas alternativas, lo cual dificulta el manejo del árbol.

5.4. Recorrido de árboles

Los tipos de recorridos son:

1) Recorridos en profundidad

Recorrido en preorden

Consiste en recorrer el nodo raíz (puede ser simplemente mostrar la clave del nodo por pantalla), después recorrer el subárbol izquierdo y una vez hecho esto, recorremos el subárbol derecho. Es un proceso recursivo por naturaleza. En el ejemplo de la Figura 26 el recorrido sería en el orden siguiente: LIBRO, TITULO, PROLOG, AUTORES, GERALD, WILFREDO.

El siguiente código es para recorrer un árbol cualquiera en preorden:

```
arbol_preorden(arbol(A,vacio,vacio),[A]) :- !.
arbol_preorden(arbol(A,X,vacio),[A|S]) :- arbol_preorden(X,S).
arbol_preorden(arbol(A,vacio,X),[A|S]) :- arbol_preorden(X,S).
arbol_preorden(arbol(A,X,Y),[A|S]) :- arbol_preorden(X,T),
                                      arbol_preorden (Y, O), append (T, O, S).
```

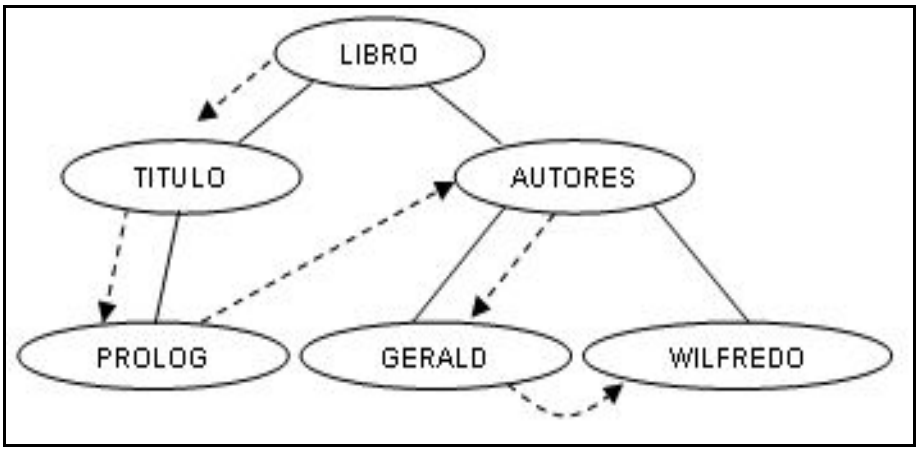


Figura 28. Recorrido de un árbol en preorden tomando el ejemplo de la figura 26.

Recorrido en inorden u orden central

Se recorre el subárbol izquierdo, el nodo raíz, y después se recorre el subárbol derecho. En el ejemplo de la Figura 26 el recorrido sería en este orden: PROLOG, TITULO, LIBRO, GERALD, AUTORES, WILFREDO.

A continuación se muestra el código para hacer el recorrido de cualquier árbol en inorden:

```
arbol_inorden(arbol(A,vacio,vacio),[A]) :- !.
arbol_inorden(arbol(A,X,vacio),S) :- arbol_inorden(X,C),append(C,[A],S).
arbol_inorden(arbol(A,vacio,X),[A|S]) :- arbol_inorden(X,S).
arbol_inorden(arbol(A,X,Y),S) :- arbol_inorden(X,C),arbol_inorden(Y,F),
                                 append(C,[A],D),append(D,F,S).
```

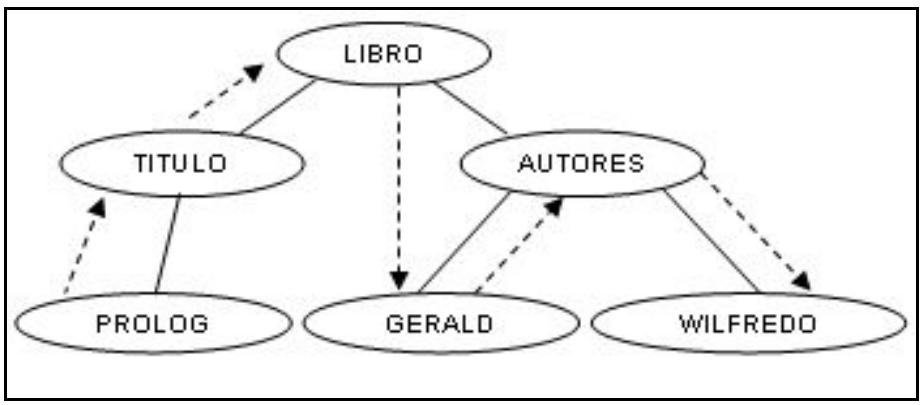


Figura 29. Recorrido de un árbol en inorden tomando el ejemplo de la figura 26.

Recorrido en postorden

Se recorre primero el subárbol izquierdo, después el subárbol derecho, y por último el nodo raíz. En el ejemplo de la figura 26 el recorrido quedaría así: PROLOG, TITULO, GERALD, WILFREDO, AUTORES, LIBRO.

El siguiente código sirve para recorrer un árbol cualquiera en postorden:

```
arbol_postorden(arbol(A,vacio,vacio),[A]) :- !.
arbol_postorden(arbol(A,X,vacio),S) :- arbol_postorden(X,C),append(C,[A],S).
arbol_postorden(arbol(A,vacio,X),S) :- arbol_postorden(X,C),append(C,[A],S).
arbol_postorden(arbol(A,X,Y),S) :- arbol_postorden(X,C),arbol_postorden(Y,F),
                                   append(C,F,D),append(D,[A],S).
```

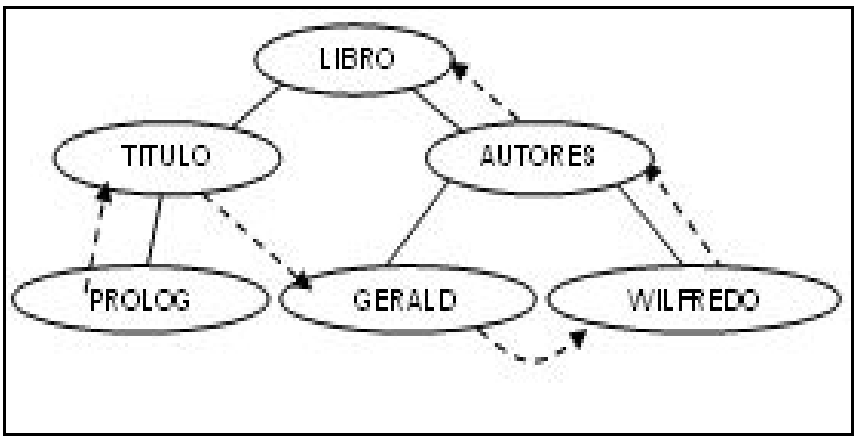


Figura 30. Recorrido de un árbol en postorden tomando el ejemplo de la figura 26.

2) Recorrido a lo ancho

En el recorrido a lo ancho se desarrollan todos los nodos del primer nivel, después todos los del segundo y así sucesivamente hasta que se examinen todos. En la siguiente figura se muestra este proceso.

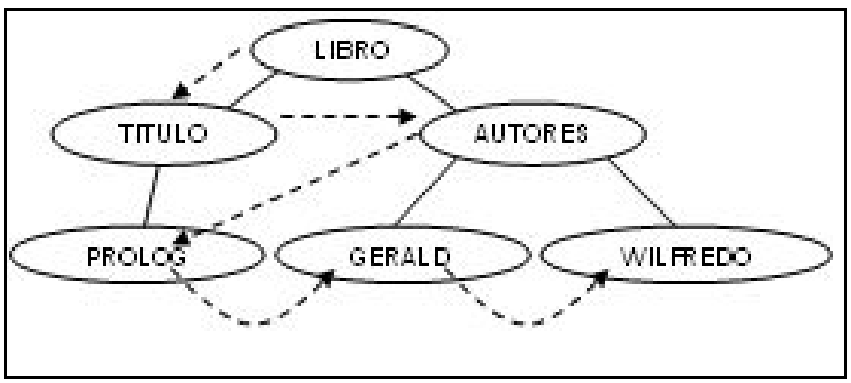


Figura 31. Recorrido de un árbol a lo ancho.

3) Recorrido heurístico

En el recorrido heurístico se utilizan metareglas o “reglas de ojo” (nos permiten elegir las reglas que aplicaremos en el recorrido del árbol) para seleccionar el siguiente nodo que hay que desarrollar.
Por ejemplo podemos elegir el nodo con la cláusula que tenga el menor elemento, o también asignar un número a las aristas entre cada par de nodos para indicar su prioridad y elegir la de mayor prioridad.

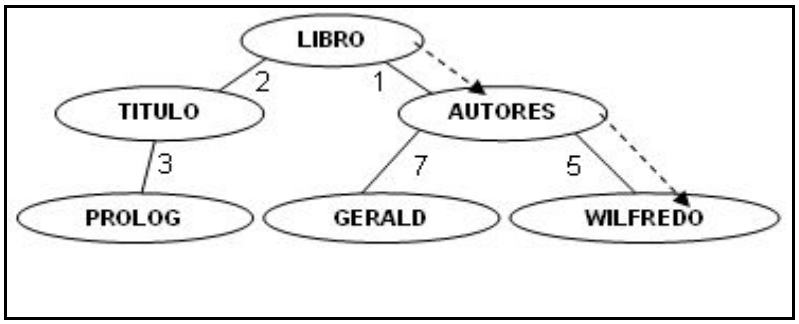


Figura 32. Recorrido heurístico de un árbol tomando el ejemplo de la figura 26. Y eligiendo los caminos de mayor prioridad.

5.5. Características del recorrido de árboles

1) La raíz del árbol es el objetivo que se quiere deducir a partir del programa.

2) En cada nivel del árbol se reemplaza el objetivo situado más a la izquierda con el cuerpo de la primera cláusula cuya cabeza unifique con dicho objetivo. Las ramas del árbol se etiquetarán con el unificador correspondiente.

3) Cuando el objetivo situado más a la izquierda fracasa se retrocede al nivel anterior y se busca otra alternativa para el objetivo situado más a la izquierda en el nivel anterior.

4) El proceso de deducción termina con éxito cuando se llegue a la cláusula vacía (no quedan más objetivos).

5) El proceso de deducción fracasa si se llega a la raíz del árbol a través del retroceso y no hay ninguna otra alternativa que explorar.

5.6. Árboles binarios

En la computación se utiliza a menudo una estructura de datos, que son los árboles binarios. Los cuales son un conjunto finito de nodos que constan de un nodo raíz que tiene dos subárboles denominados subárbol izquierdo y subárbol derecho. Estos árboles tienen 0, 1 ó 2 descendientes como máximo. El árbol de la Figura 26, es un ejemplo válido de árbol binario.

5.6.1. Árboles binarios de búsqueda

Una de las principales aplicaciones de los árboles es que permiten almacenar datos en ellos, lo cual hace que la búsqueda de un determinado elemento resulte muy eficiente.

Cuando se usan árboles binarios, la búsqueda se puede realizar de forma más sencilla, ya que cada vez que se alcanza un determinado nodo, solamente es necesario inspeccionar uno de los subárboles.

En un árbol binario de búsqueda las ramas de cada nodo están ordenadas y dicho árbol puede ser:

1) Una estructura vacía o
2) Un elemento o clave de información (nodo) más un número finito (como máximo dos) de estructuras de tipo árbol. Dichos nodos están disjuntos, y se les denomina subárboles, además cumplen lo siguiente:

- **a.** Todas las claves del subárbol izquierdo al nodo son menores que la clave del nodo.
- **b.** Todas las claves del subárbol derecho al nodo son mayores que la clave del nodo.
- **c.** Ambos subárboles son árboles binarios de búsqueda.

Un ejemplo de árbol binario de búsqueda:

Utilizando la siguiente regla:

arbol(arbol(4,arbol(2,arbol(1,vacio,vacio),arbol(3,vacio,vacio)),
arbol(6,arbol(5,vacio,vacio),árbol(7,vacio,vacio))).

El árbol binario de búsqueda sería:

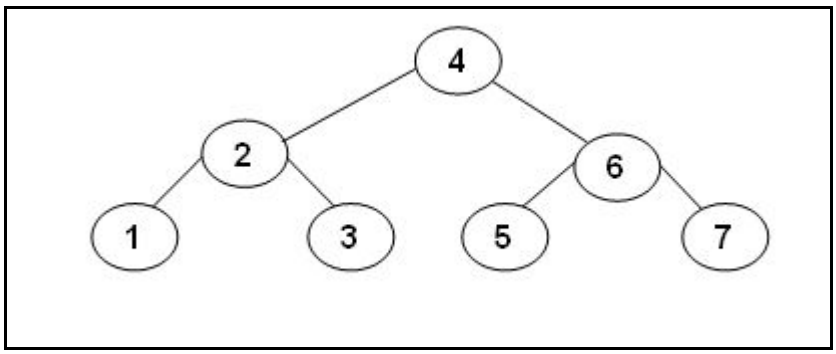


Figura 33. Ejemplo de árbol binario de búsqueda.

Una ventaja fundamental de los árboles de búsqueda es que por lo general son mucho más rápidos para localizar un elemento que en una lista. Por tanto, son más rápidos para insertar y borrar elementos.

5.6.2. Operaciones básicas sobre árboles binarios de búsqueda

Búsqueda: Aprovechando las propiedades del árbol de búsqueda se puede acelerar la localización. Simplemente hay que descender a lo largo del árbol por la izquierda o bien por la derecha dependiendo del elemento que se busca.

Por ejemplo, si en el árbol de la Figura 33, queremos encontrar el nodo con el elemento número 3, primero descendemos por la izquierda del nodo raíz y

llegaríamos al nodo con el elemento número 2, luego descenderíamos por la derecha de este nodo y de esta forma encontraríamos el nodo a buscar.

Inserción: Es prácticamente idéntica a la búsqueda, es decir que se desciende a lo largo del árbol ya sea por la izquierda o por la derecha dependiendo de donde vayamos a insertar el elemento. Cuando se llega a un árbol vacío se crea un nodo, de esta manera los nuevos enlaces mantienen la coherencia. Si el elemento a insertar ya existe entonces no se hace nada.

Borrado: La operación de borrado resulta un poco más complicada. El árbol debe seguir siendo de búsqueda y pueden darse tres casos, una vez encontrado el nodo a borrar:

1) El nodo no tiene descendientes. Simplemente se borra.

2) El nodo tiene al menos un descendiente por una sola rama. Se borra dicho nodo, y su primer descendiente se asigna como hijo del padre del nodo borrado.

3) El nodo tiene al menos un descendiente por cada rama. Al borrar dicho nodo es necesario mantener la coherencia de los enlaces, además de seguir manteniendo la estructura como un árbol binario de búsqueda. La solución consiste en sustituir la información del nodo que se borra por el de una de las hojas, y borrar a continuación dicha hoja. No puede ser cualquier hoja, debe ser la que contenga una de estas dos claves:

- La mayor de las claves menores al nodo que se borra. Suponer que se quiere borrar el nodo 2 del árbol de la Figura 33,. Se sustituirá la clave 2 por la clave 1.

- La menor de las claves mayores al nodo que se borra. Suponer que se quiere borrar el nodo 4 del árbol de la Figura 33,. Se sustituirá la clave 2 por la clave 3.

Ejemplo: en el árbol de la Figura 33, se borra el nodo cuya clave es 2 (tomamos el caso del inciso b, la clave 2 es sustituida por la clave 3). El árbol resultante es:

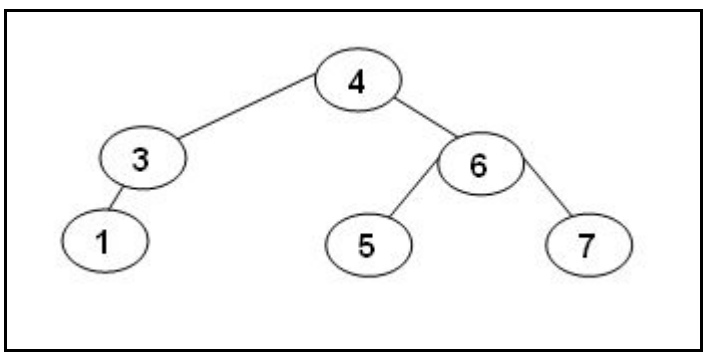


Figura 34. Árbol resultante al eliminar el nodo de clave 2 en la Figura 33.

5.7. Ejercicios resueltos

1) Dada la siguiente regla:

arbol(libro,arbol(titulo,arbol(prolog,vacio,vacio),vacio),
arbol(autores,arbol(gerald,vacio,vacio),arbol(wilfredo,vacio,vacio))).

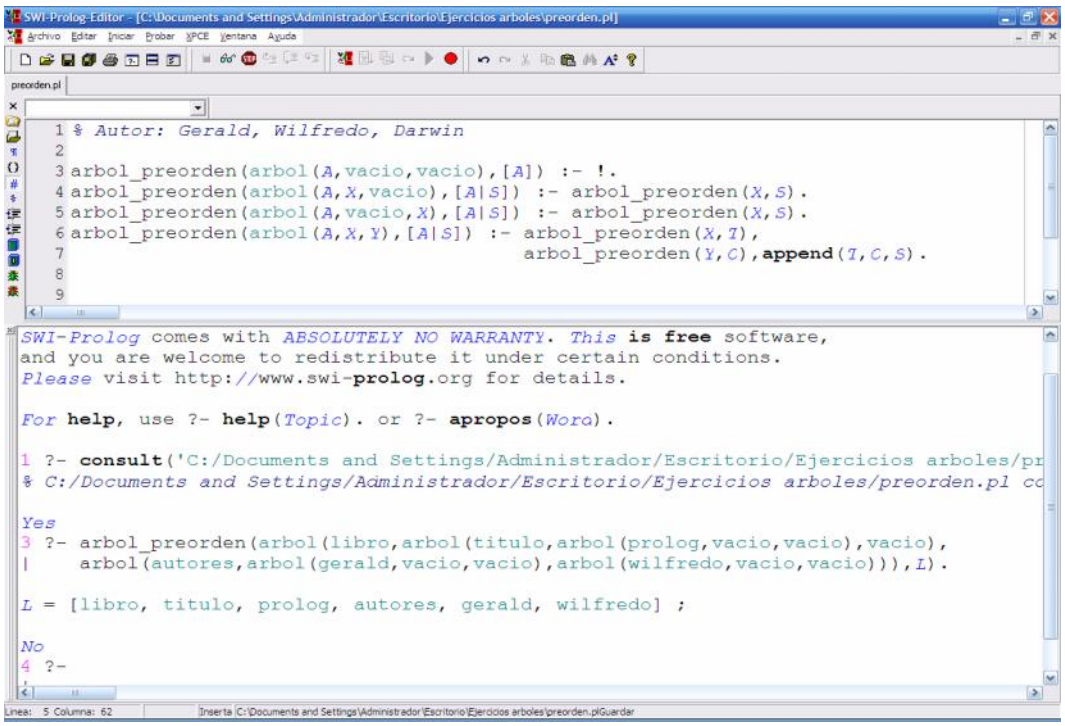
Desarrolle los predicados correspondientes para recorrer el árbol de la Figura 26, en:

a) preorden
b) inorden
c) postorden

Mostrando el resultado en una lista L.

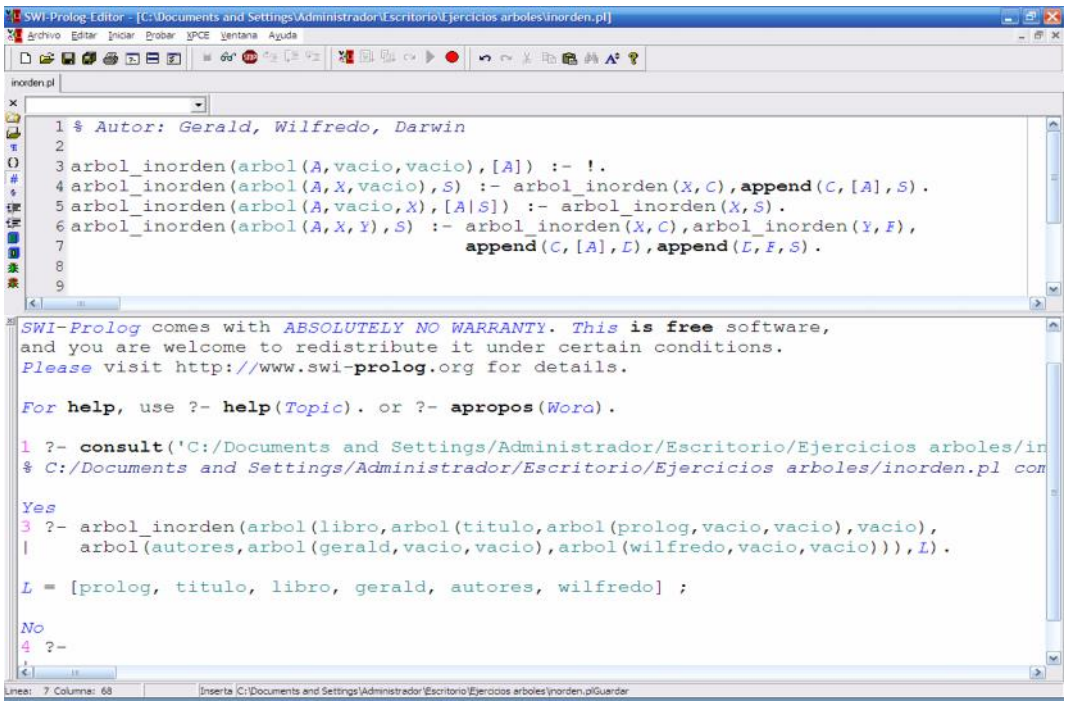
a) Árbol en preorden:

Solución:



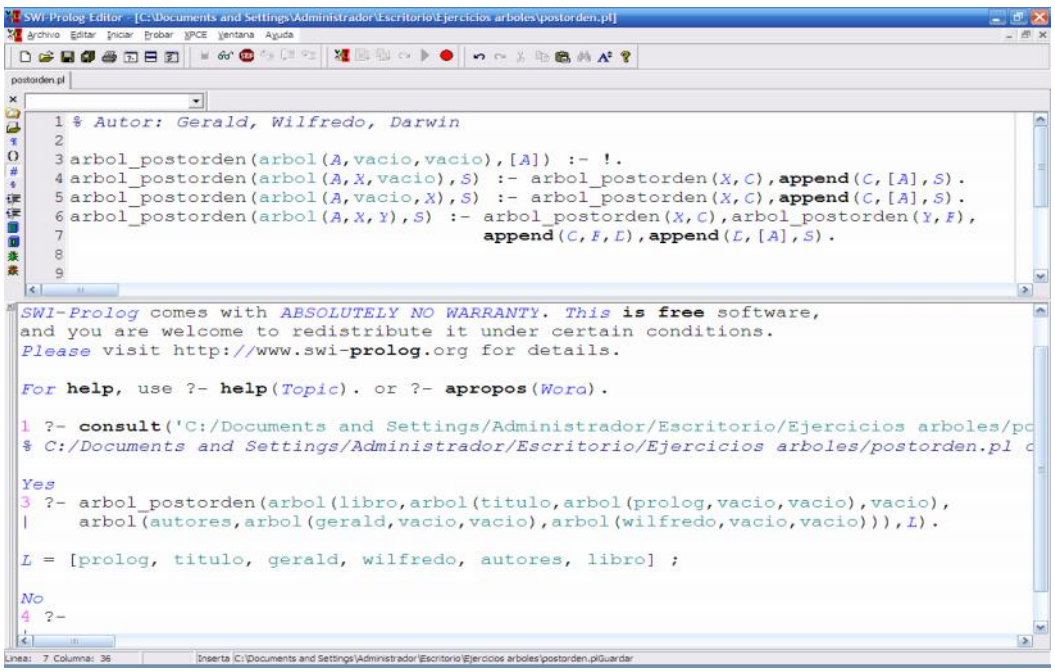
b) Árbol en inorden:

Solución:



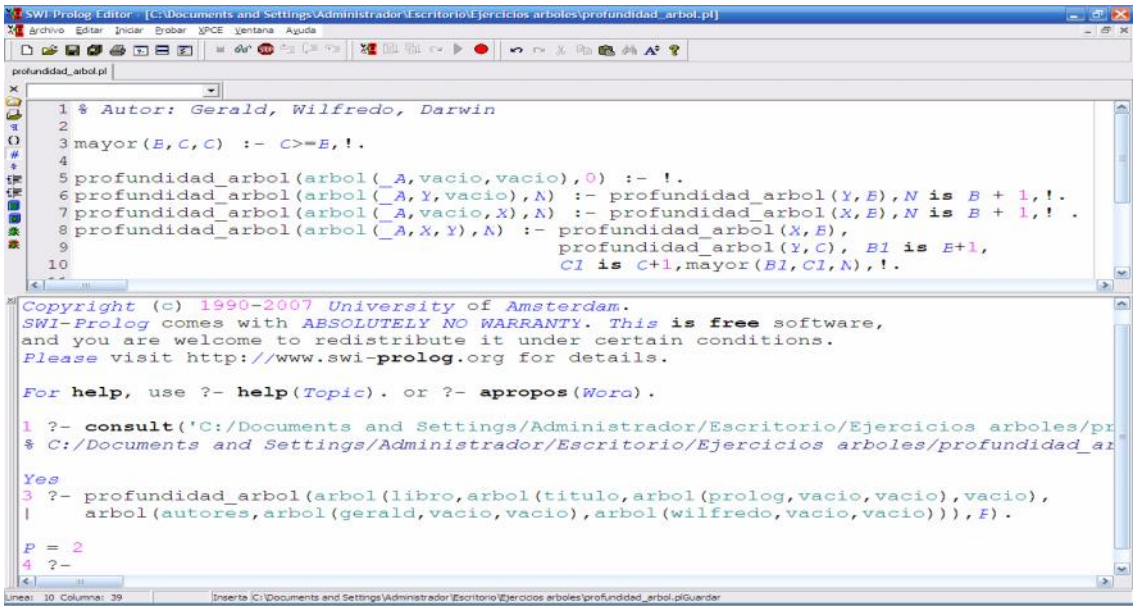
b) Árbol en postorden:

Solución:

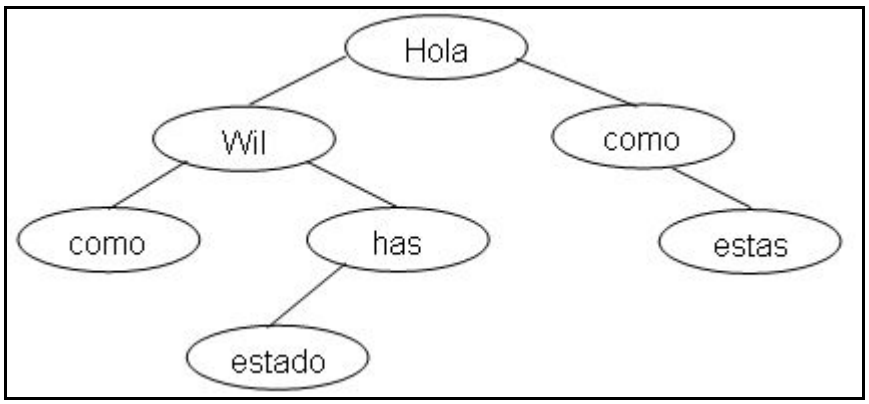


2) Elabore un programa que calcule la profundidad del árbol de la figura 26, Donde la raíz tiene nivel 0.

Solución:

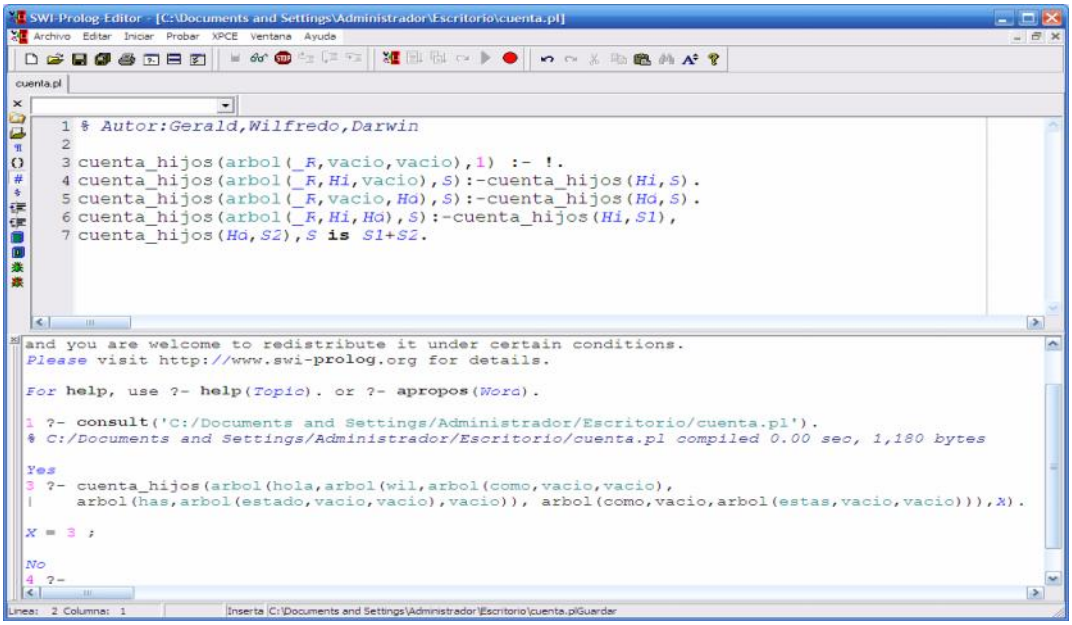


3) Elabore un programa que devuelva el número de hijos del siguiente árbol (para árboles con solo la raíz, se cuenta un hijo, que es la misma raíz).



?-cuenta_hijos(arbol(hola,arbol(wil,arbol(como,vacio,vacio),
arbol(has,arbol(estado,vacio,vacio),vacio)),
arbol(como,vacio,arbol(estas,vacio,vacio))),X).

Solución:


SWI-Prolog-Editor - [C:\Documents and Settings\Administrador\Escritorio\cuenta.pl]
Archivo Editar Iniciar Probar XPCE Ventana Ayuda
cuenta.pl
1 % Autor:Gerald,Wilfredo,Darwin
2
3 cuenta_hijos(arbol(_R,vacio,vacio),1) :- !.
4 cuenta_hijos(arbol(_R,Hi,vacio),S):-cuenta_hijos(Hi,S).
5 cuenta_hijos(arbol(_R,vacio,Hd),S):-cuenta_hijos(Hd,S).
6 cuenta_hijos(arbol(_R,Hi,Hd),S):-cuenta_hijos(Hi,S1),
7 cuenta_hijos(Hd,S2),S is S1+S2.
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
1 ?- consult('C:/Documents and Settings/Administrador/Escritorio/cuenta.pl').
% C:/Documents and Settings/Administrador/Escritorio/cuenta.pl compiled 0.00 sec, 1,180 bytes
Yes
3 ?- cuenta_hijos(arbol(hola,arbol(wil,arbol(como,vacio,vacio),
| arbol(has,arbol(estado,vacio,vacio),vacio)), arbol(como,vacio,arbol(estas,vacio,vacio))),X).
X = 3 ;
No
4 ?-
Linea: 2 Columna: 1
Inserta C:\Documents and Settings\Administrador\Escritorio\cuenta.plGuardar

Tema 6: Programación lógica y base de datos.

En este tema pretendemos presentar al estudiante los principales aspectos relacionados con la programación lógica y base de datos. Hablaremos del algebra relacional y sus diferentes operaciones que en otros lenguajes son tratadas como tablas y en Prolog se conocen como hechos.

Contenido:

6. Programación lógica y base de datos.
6.1 Definición de base de datos.
6.2 Tipos de base de datos.
6.2.1 Base de Datos Lógicas.
6.2.2 Base de Datos Orientadas a Objetos.
6.2.3 Base de Datos Relacionales.
6.3 Álgebra Relacional.
6.4 Representación de las relaciones en Prolog.
6.5 Operaciones fundamentales.
6.5.1 Selección.
6.5.2 Proyección.
6.5.3 Unión.
6.5.4 Diferencia de conjuntos.
6.5.5 Producto cartesiano.
6.5.6 Intersección de conjuntos.
6.5.7 La operación reunión natural
6.5.8 División
6.6 Ejercicios resueltos.

6. Programación lógica y base de datos

6.1. Definición de base de datos

Una base de datos es un sistema para archivar información en una computadora cuyo propósito general es mantenerla y hacer que esté disponible cuando se solicite. Las bases de datos son un área de la computación que ha recibido mucha atención debido a sus múltiples aplicaciones: bibliotecas, automatización de oficinas, ingeniería de software, diccionarios automatizados y en general cualquier programa orientado a mantener y recuperar información textual. Su recuperación, actualización y manejo es relativamente simple con el uso de cualquier manejador de bases de datos.

6.2. Tipos de base de datos

Entre los tipos de base de datos tenemos:

- Bases de Datos Lógicas.
- Bases de Datos Orientadas a Objetos.
- Base de Datos Relacionales.

6.2.1. Bases de datos lógicas

Las bases de datos lógicas son construidas con registros homogéneos de manera parecida a las relacionales. Adicionalmente se agregan restricciones lógicas y reglas de composición parecidas a las de Prolog, que permiten en principio deducir información que originalmente no está contenida en la base de datos.

6.2.2. Bases de datos orientadas a objetos

Las bases de datos orientadas a objetos, tienen una organización similar a la de los árboles. Donde cada nodo del árbol representa un campo y cada árbol un registro, cada tipo de nodo tiene un método distinto de búsqueda. Que es equivalente a decir que todos los campos pueden ser utilizados como campos llave, pero complica el diseño. Si la base de datos es demasiado grande, o tiene relaciones demasiado complejas, el grafo resultante se vuelve complicado. Actualmente no existen implementaciones comerciales de este tipo de bases de datos.

6.2.3. Bases de datos relacionales

En los esquemas tradicionales, las bases de datos relacionales son las que han tenido más uso comercial, ya que el modelo relacional se ha establecido como el principal modelo de datos para las aplicaciones de procesamiento de datos.

Una base de datos relacional consiste en un conjunto de tablas o relaciones, en la que cada fila (o tupla) de la tabla representa una relación entre un conjunto de valores. Y cada columna de la tabla es un atributo. Para cada atributo hay un conjunto de valores permitidos, correspondiendo al dominio de dicho atributo. Cada relación tendrá una clave primaria, que será un subconjunto de sus atributos, y cuyo valor será único dentro de la misma relación. A través de la clave primaria se podrá hacer referencia a cada una de las tuplas de la relación de manera unívoca.

6.3. Algebra relacional

El algebra relacional es un lenguaje de consulta procedimental. Consta de un conjunto de operaciones que toman como entrada una o dos relaciones y producen como resultado una nueva relación.

6.3.1. Representación de las relaciones en Prolog

Los siguientes datos representados mediante tablas en las bases de datos relacionales se pueden mostrar en Prolog mediante un conjunto de hechos.

Ejemplo clásico de suministradores y partes.

Suministradores			
Scodigo	**Snombre**	**estatus**	**Ciudad**
s1	wilfredo	21	managua
s2	alicia	22	chinandega
s3	nubia	25	león
s4	angelita	21	esteli
s5	darwin	25	chinandega

Partes				
Pcodigo	**Pnombre**	**Color**	**Peso**	**Ciudad**
p1	cama	blanco	18	chinandega
p2	reloj	blanco	4	chinandega
p3	alfombra	rojo	15	león
p4	silla	café	6	esteli
p5	sofá	marrón	18	managua
p9	televisor	negro	50	chinandega

S_p		
Scodigo	**Pcodigo**	**Cantidad**
s1	p1	400
s1	p2	100
s1	p3	300
s1	p4	400
s1	p5	800
s1	p6	200
s2	p1	400
s2	p2	300
s3	p2	100
s4	p2	100
s4	p4	400
s4	p5	300

Estas tablas de la base de datos relacional del ejemplo de los suministradores y partes pueden ser representadas en Prolog con los siguientes hechos:

```
suministradores (s1, wilfredo, 21, managua).
suministradores (s2, alicia, 22, chinandega).
suministradores (s3, nubia, 25, león).
suministradores (s4, angela, 26, esteli).
suministradores (s5, darwin, 25, chinandega).
```

```
partes (p1, cama, blanco, 18, chinandega).
partes (p2, reloj, blanco, 4, chinandega).
partes (p3, alfombra, rojo, 15, león).
partes (p4, silla, café, 6, esteli).
partes (p5, sofá, marrón, 18, managua).
partes (p6, televisor, negro, 50, chinandega).
```

```
s_p (s1, p1, 400).
s_p (s1, p2, 100).
s_p (s1, p3, 300).
s_p (s1, p4, 400).
s_p (s1, p5, 800).
s_p (s1, p6, 200).
s_p (s2, p1, 400).
s_p (s2, p2, 300).
s_p (s3, p2, 100).
s_p (s4, p2, 100).
s_p (s4, p4, 400).
s_p (s4, p5, 300).
```

6.3.2. Operaciones fundamentales

Las operaciones fundamentales del algebra relacional son selección, proyección, unión, diferencia de conjuntos, productos cartesiano y renombramiento. Además de estas operaciones elementales hay otras operaciones, como la intersección de conjuntos, la reunión natural, la división y la asignación, que se definen en términos de las operaciones fundamentales.

Estas operaciones que permiten generar nuevas relaciones a partir de las existentes, pueden ser reproducidas fácilmente en Prolog como se muestra a continuación.

6.3.2.1. La operación de selección

En algebra relacional

Es denotada por la letra griega sigma minúscula **σ**, y selecciona tuplas que satisfacen un predicado dado.

Por ejemplo: para seleccionar las tuplas de la relación suministradores en que la ciudad es “chinandega” hay que escribir:

σciudad=chinandega (suministradores)

Para seleccionar se pueden utilizar los operadores de comparación **=, ≠,>, ≥, <, ≤** y también se pueden combinar varios predicados con las conectivas ∧,∨**.**

En Prolog:

La selección se consigue utilizando la coincidencia sintáctica del Prolog. Es decir cuando una relación tiene su clave primaria, la cual será un subconjunto de sus atributos, y cuyo valor es único dentro de la misma relación. A través de esta se podrá coincidir sintácticamente cada una de las tuplas de la relación de manera unívoca.

Por ejemplo: Proporcionar el nombre de los suministradores de chinandega con estatus mayor que 20.

```
consulta(Snombre):-suministradores(_,Snombre,Estatus,chinandega),
Estatus>20.
```

6.3.2.2. La operación proyección

En algebra relacional

La proyección es denotada por la letra griega mayúscula pi (π) y selecciona columnas o atributos de una relación, eliminando las filas duplicadas.

Ejemplo: Obtener el nombre y el código de todos los suministradores.

$$\pi_{\text{scodigo,snombre}}(\text{suministradores})$$

También se puede hacer composición de operaciones. Como por ejemplo para obtener el nombre y el código de aquellos suministradores de la ciudad de chinandega hay que escribir:

$$\pi_{\text{scodigo, snombre}}(\sigma_{\text{ciudad= chinandega}}(\text{suministradores}))$$

En Prolog

La proyección se consigue escribiendo como argumento(s) de la cabeza de la regla que definiría la consulta, el nombre del atributo(s) que interesa proyectar.

Ejemplo: Proyectar el nombre de los suministradores de chinandega con estatus mayor que 20.

```
proyectar(Snombre):-suministradores(_,Snombre,Estatus,chinandega),
Estatus>20.
```

6.3.2.3. La operación de unión

En algebra relacional

La operación es denotada como en la teoría de conjuntos por **U** y permite unir las tuplas de dos relaciones iguales (misma aridad y los dominios de los atributos de ambas relaciones tienen que ser iguales), eliminando tuplas repetidas.

Ejemplo: Obtener el nombre de los suministradores de la ciudad de chinandega o de la ciudad de esteli.

π **snombre(σciudad= chinandega (suministradores)) U**

π **snombre(σciudad= esteli(suministradores))**

En Prolog

La unión se obtiene definiendo predicados con el mismo nombre que resuelvan cada una de las relaciones que se deben unir (es decir, utilizando la **v** lógica del Prolog).

Ejemplo: Para obtener el nombre de los suministradores de la ciudad de chinandega o de la ciudad de valencia, se escribirá en Prolog:

```
consulta(Snombre):- suministradores (_,Snombre,_, chinandega ).
consulta(Snombre):- suministradores (_,Snombre,_,valencia).
```

6.3.2.4. La operación diferencia de conjuntos

En algebra relacional

Es denotada por - y permite obtener las tuplas de una relación que no están en la otra. Ambas relaciones tienen que ser iguales.

Ejemplo: Obtener los códigos de los suministradores que no han suministrado ninguna parte.

π **scodigo (suministradores)** - π **. scodigo (s_p)**

En Prolog

La operación de diferencia de conjuntos se obtiene utilizando el predicado predefino "**not**".

Ejemplo: Obtener el código de los suministradores que no hayan suministrado ninguna parte.

```
consulta(Scodigo):- suministradores (Scodigo,_,_,_),not  s_p(Scodigo,_,_).
```

6.3.2.5. La operación producto cartesiano

En algebra relacional

La operación producto cartesiano denotada por un aspa (**x**), permite combinar información de dos relaciones. Para distinguir los atributos con el mismo nombre en las dos relaciones, se pone como prefijo el nombre de la relación de donde provienen, seguidos de un punto.

Ejemplo: Para obtener el nombre de los suministradores de la ciudad de chinandega que hayan suministrados partes alguna vez.

π **snombre(σsuministradores.scodigo=s_p. scodigo(σciudad= chinandega (suministradores x partes)))**

El producto cartesiano (suministradores x partes) asocia todas las tuplas de suministradores con todas las tuplas de partes. Después se hace una selección de aquellas tuplas cuya ciudad es chinandega, y posteriormente se hace una selección de las tuplas en las que suministradores.scodigo=s_p.scodigo. Por último se hace una proyección del atributo Snombre.

Más eficiente resulta hacer el producto cartesiano del resultado de la selección de aquellas tuplas que nos interesan.

π snombre(σ suministradores.scodigo=s_p. scodigo((σciudad= chinandega (suministradores)) x partes)))

En Prolog

La operación producto cartesiano se obtiene mediante la coincidencia sintáctica.

Ejemplo: Obtener los nombres de los suministradores de la ciudad de chinandega que hayan suministrados partes algunas vez.

```
consulta(Snombre):-suministradores(Scodigo,Snombre,_,chinandega),
s_p(Scodigo,_,_).
```

6.3.2.6. La operación intersección de conjuntos

En algebra relacional

La operación intersección de conjuntos es denota ∩, y permite interceptar las tuplas de relaciones iguales.

Ejemplo: Obtener los nombres de los suministradores de chinandega y de esteli.

π snombre(σciudad=chinandega(suministradores)) ∩

π snombre(σciudad=esteli(suministradores))

En Prolog

La operación intersección de conjuntos se obtiene utilizando también la coincidencia sintáctica.

Ejemplo: Obtener el nombre de los suministradores que han suministrados la pieza con código “p1” y la pieza “p2”.

```
consulta(Snombre):-suministradores(Scodigo,Snombre,_,_),
s_p(Scodigo,p1,_), s_p(Scodigo,p2,_).
```

6.3.2.7. La operación reunión natural

En Algebra Relacional

La reunión natural es denotada por el símbolo de la reunión □, permite combinar ciertas selecciones y un producto cartesiano en una sola operación. Esta operación lo que hace es formar un producto cartesiano de sus dos argumentos, realizar una selección forzando la igualdad de los atributos que aparecen en ambos esquemas de la relación y finalmente eliminar los atributos duplicados.

Ejemplo: Para obtener el nombre de los suministradores de la ciudad de chinandega que hayan suministrado partes alguna vez.

π **snombre ((σciudad= chinandega (suministradores)) □ partes))**

6.3.2.8. La operación de división

En Algebra Relacional

La operación de división es denotada por ÷, y resulta adecuada para las consultas que incluyen la expresión “para todos”.

Ejemplo: Obtener los suministradores que suministran todas las partes.

r1 = π **scodigo,pcodigo(s_p)**

r2 = π **pcodigo(partes)**
r3 = r1 ÷ r2

r4= π **scodigo(r3 x suministradores)**

En Prolog

La operación de división se obtiene utilizando el predicado predefinido “findall”, el cual recoge en una lista todas las instancias de una variable en un objetivo. Es decir supongamos que necesitamos generar todas las soluciones para un objetivo dado y además, disponer de ellas agrupadas dentro de una estructura. Para esto, Prolog dispone de este predicado.

Ejemplo: Para obtener el nombre de los suministradores que suministran todas las partes.

```
consulta(Snombre):- findall(Pcodigo,partes(Pcodigo,_,_,_,_),L_todas),
suministradores(Scodigo,Snombre,_,_),
findall(Pcodigo,s_p(Scodigo,Pcodigo,_),L_ suministradores),
subcto_partes (L_todas, L_ suministradores).
Subcto_partes ([], _).
subcto_partes ([H|T], Lista):- miembro (H, Lista), subcto_partes (T, Lista).
```

6.4. Ejercicios resueltos

Ejercicio 1

Unos clubes desean desarrollar una Base de Datos en la cual se podrán definir relaciones como:

competición(P#,DESCRIPCION,CATEGORIA).
club(C#,NOMBRE_C,PRESUPUESTO).
participación(C#, P#,PUESTO).

En donde la relación competición representa los atributos que describen las características del tipo de competencia y categoría a la que podrán optar los participantes de los diferentes clubes. La relación club posee los atributos que describen los nombres de los distintos clubes, cada uno con la cantidad de presupuestó con la que cuenta. Ambas relaciones se instancia por medio de la relación participación a través de sus llave primarias y describe el lugar que ocupa al finalizar cada participante en la competencia.

```
competicion(p1,nochedemariachi,1).
competicion(p1,nochedemariachi,2).
competicion(p2,nochedemariachi,1).
competicion(p2,nochedeestrella,2).
competicion(p3,nochedemariachi,3).
competicion(p4,nochedemariachi,4).
competicion(p5,nochedeestrella,2).
```

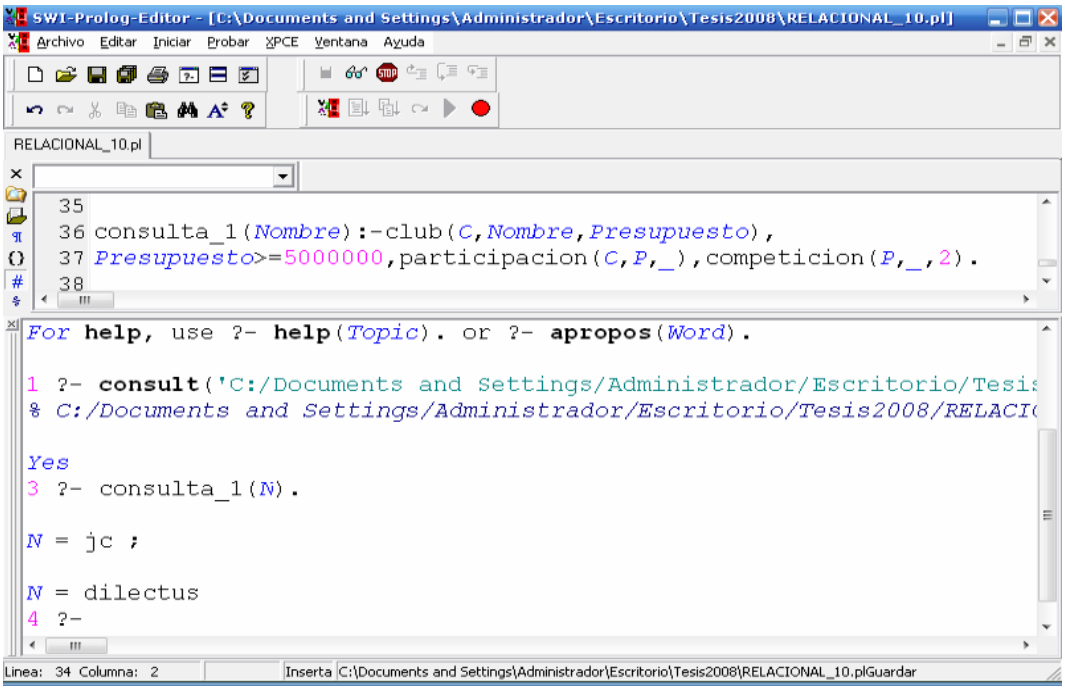
```
club(c1,jc,5000000).
club(c2,genesis,4000000).
club(c3,starclub,3000000).
club(c4,almendro,2000000).
club(c5,dilectus,7000000).
```

```
participacion(c1,p1,primero).
participacion(c2,p2,primero).
participacion(c3,p3,segundo).
participacion(c4,p4,tercero).
participacion(c5,p5,segundo).
```

Se pide dar respuesta a las siguientes consultas en Prolog:

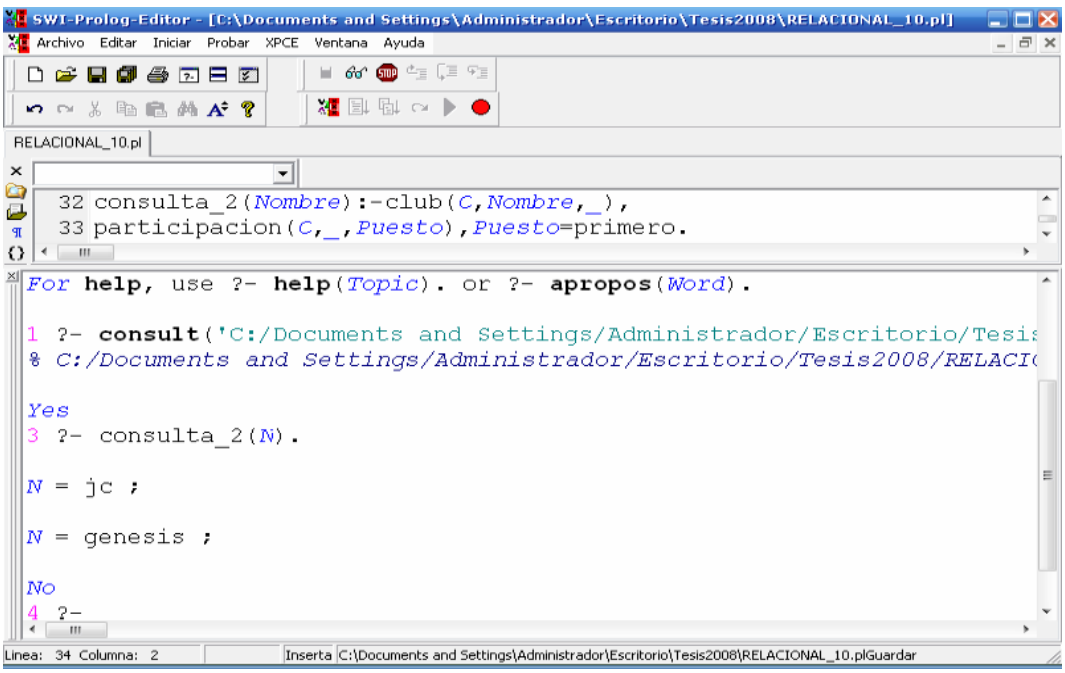
1) Obtener los nombres de los clubes con presupuesto mayor o igual que 5 millones y que hayan participado en competiciones de categoría igual a 2.

Solución:



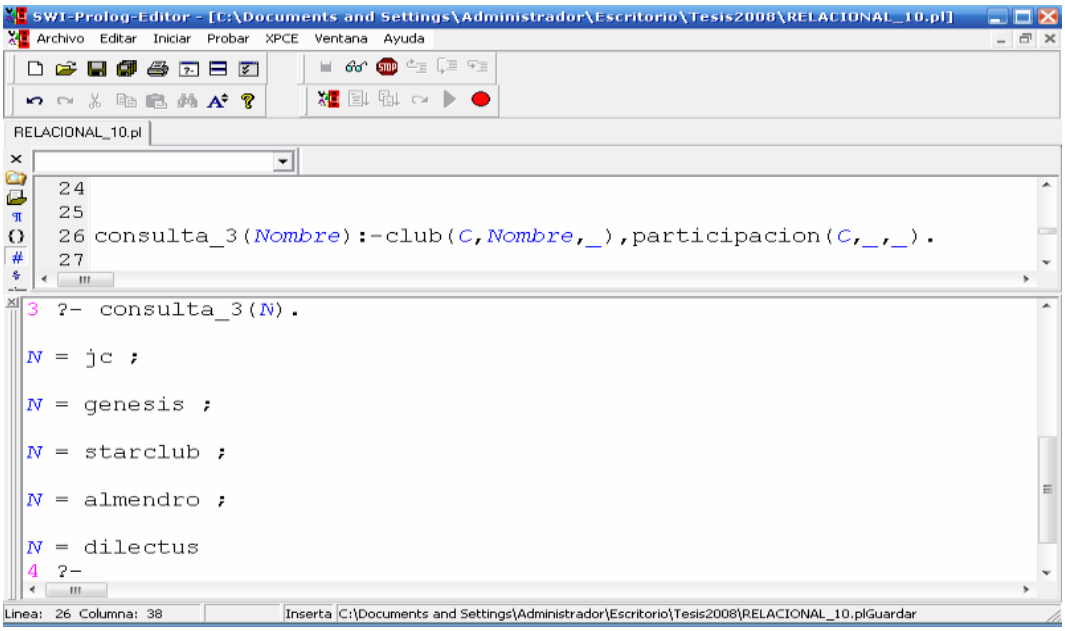
2) Obtener los nombres de los clubes que solo han conseguido el primer puesto.

Solución:



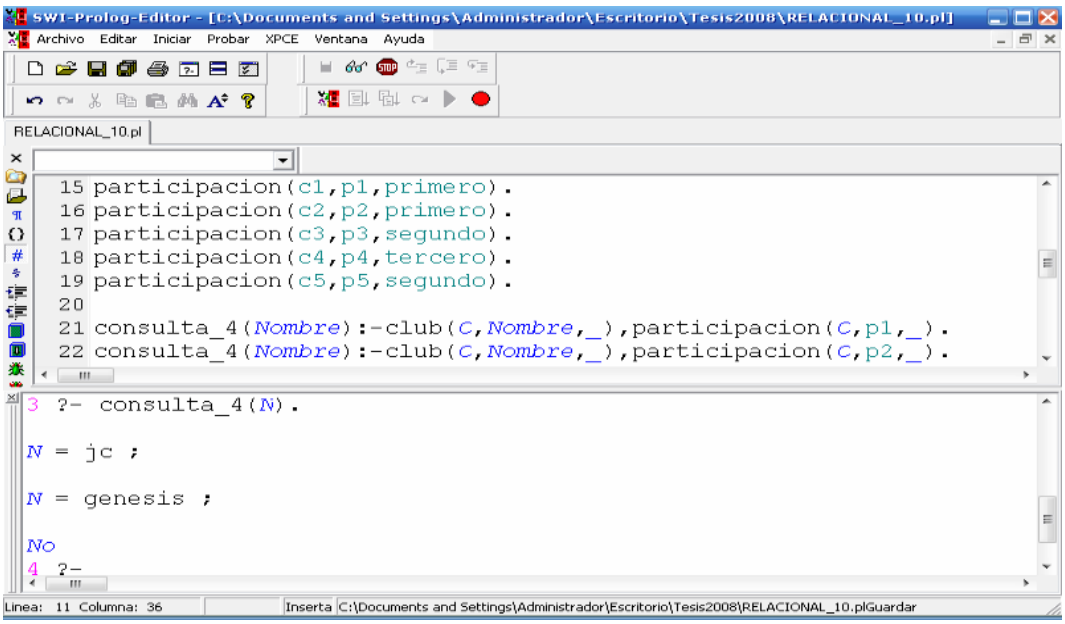
3) Obtener los nombres de los clubes que han participado en todas las competiciones.

Solución:



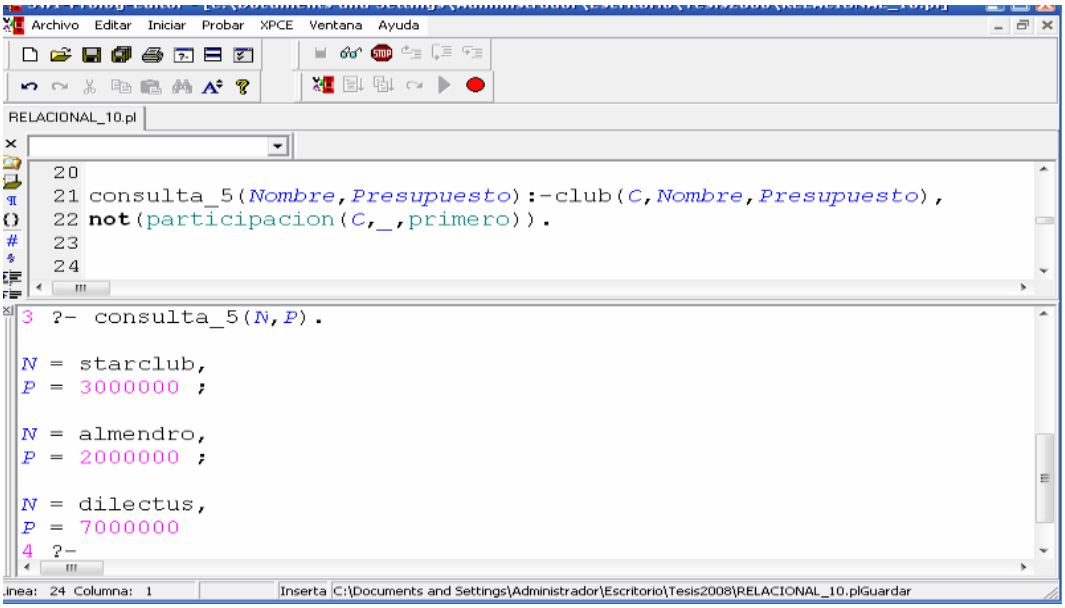
4) Obtener los nombres de los clubes que han participado en las competiciones con código P1 o código P2.

Solución:



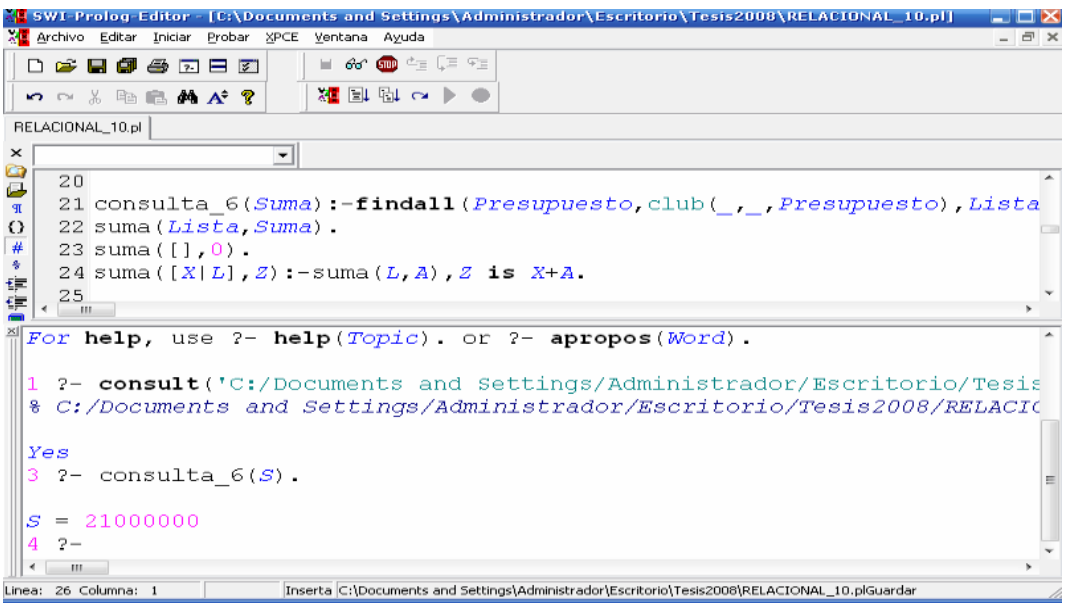
5) Obtener el nombre y presupuesto de los clubes que no han conseguido un primer puesto.

Solución:



6) Proyectar la cantidad total del presupuesto de todos los clubes.

Solución:



Ejercicio 2

Dada una Base de Dato Relacional, cuyas relaciones son:

autor(DNI#,NOMBRE,UNIVERSIDAD).
tema(COD_TEMA,N_Revista).
revista(REFERENCIA,TITULO_REV,EDITORIAL).
articulo(REFERENCIA,DNI#,COD_TEMA,TITULO_ART,AÑO,VOLUMEN,NUMERO,PAGINAS).

En donde cada relación describe las características pertenecientes a un grupo de escritores de artículos de diversos temas de la vida cotidiana. La relación autor posee los atributos nombre y la universidad a la que pertenece cada escritor, la relación tema contiene el código del tema y el nombre de la revista, la relación revista contiene la referencia de los distintos artículos y la editorial de publicación, los cuales se relacionan por medio artículo el cual contienes sus distintas llaves primarias, además del año de publicación de cada artículo, la cantidad de volumen y el número de páginas publicadas.

```
autor(01,pablo,unan_leon).
autor(02,jose,ucc).
autor(03,gioconda,ucan).
autor(04,felipe,unan_leon).
autor(05,marcos,unan_leon).
```

```
tema(esp,espanol).
tema(mat,matematica).
tema(prog,programacion).
tema(bio,biologia).
tema(eco,economia).
tema(acui,acuicola).
tema(finz,inflacion).
```

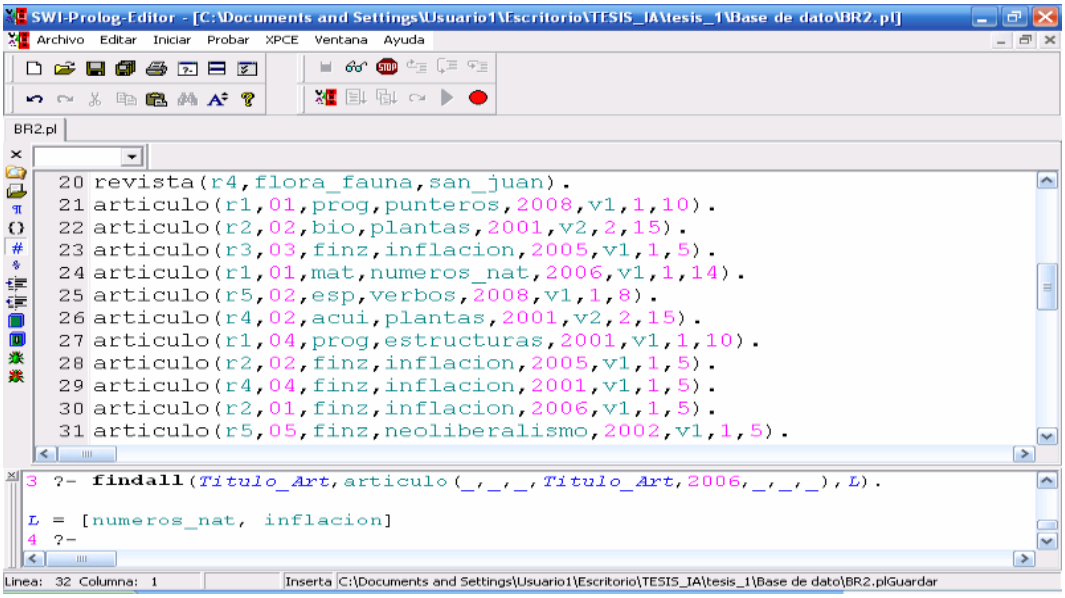
```
revista(r1,informatica,rama).
revista(r2,seres_vivos,hispan).
revista(r3,globalizacion,san_juan).
revista(r1,numeros_primos,rama).
revista(r5,la_oracion,hispan).
revista(r4,flora_fauna,san_juan).
```

```
articulo(r1,01,prog,punteros,2008,v1,1,10).
articulo(r2,02,bio,plantas,2001,v2,2,15).
articulo(r3,03,finz,inflacion,2005,v1,1,5).
articulo(r1,01,mat,numeros_nat,2006,v1,1,14).
articulo(r5,02,esp,verbos,2008,v1,1,8).
articulo(r4,02,acui,plantas,2001,v2,2,15).
articulo(r1,04,prog,estructuras,2001,v1,1,10).
articulo(r2,02,finz,inflacion,2005,v1,1,5).
articulo(r4,04,finz,inflacion,2001,v1,1,5).
articulo(r2,01,finz,inflacion,2006,v1,1,5).
articulo(r5,05,finz,neoliberalismo,2002,v1,1,5).
```

Se pide dar respuesta algebraica a las siguientes consultas en Prolog:

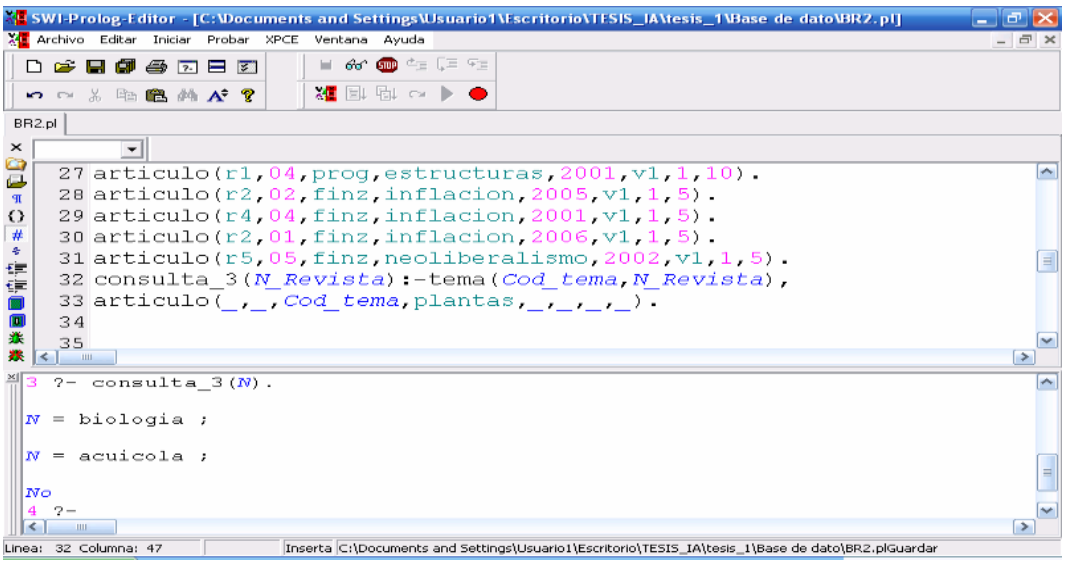
1) Obtener los artículos cuyo titulo hayan sido publicados en el año 2006.

Solución:



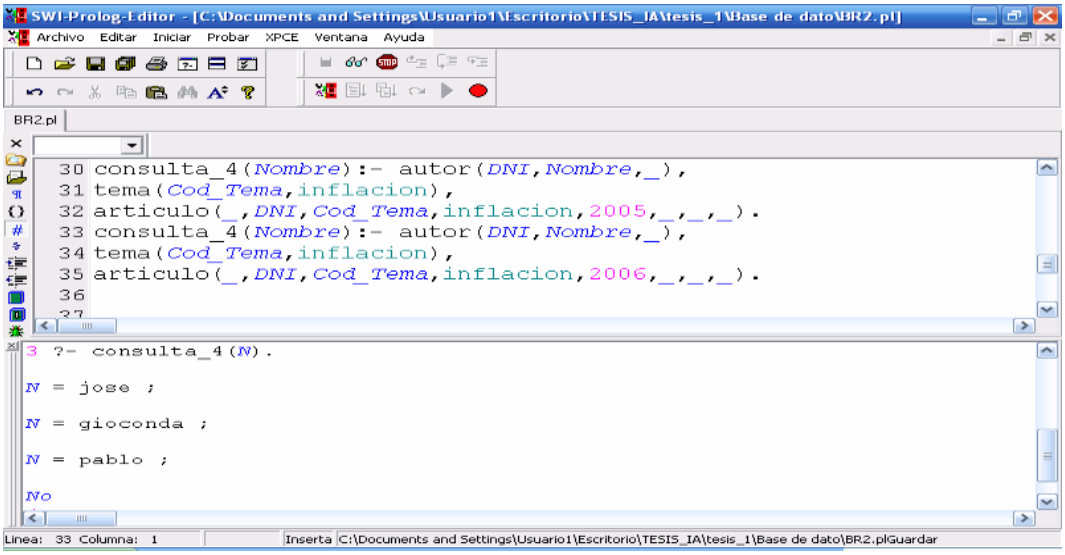
2) Obtener las revistas que solo publican artículos cuyo tema sea sobre plantas.

Solución:



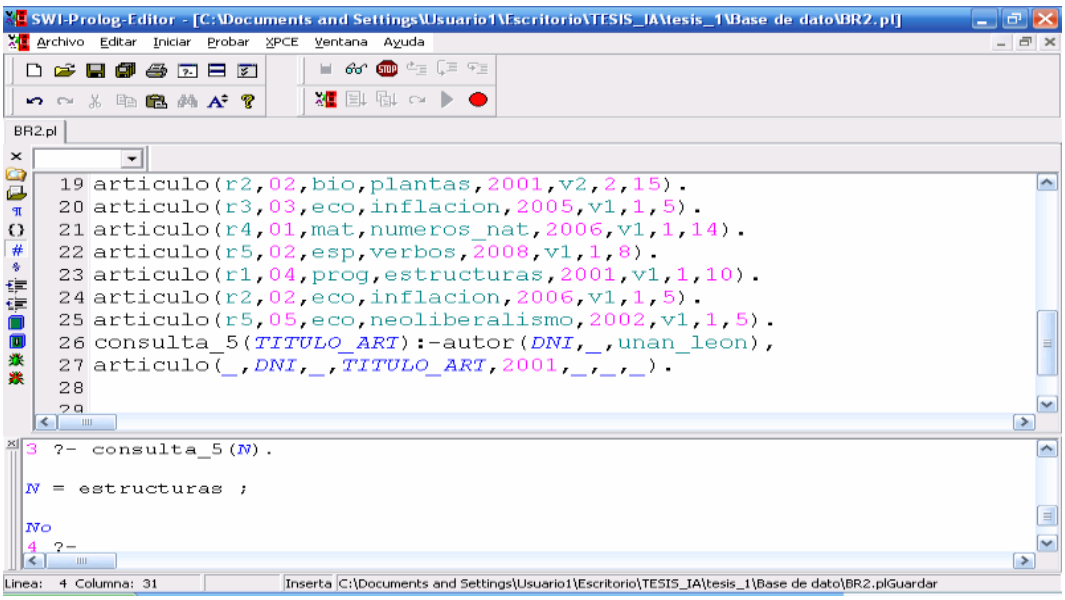
3) Obtener los autores que han publicado artículos del tema inflación, en el año 2005, o en el año 2006.

Solución:



4) Obtener los artículos del año 2001 publicados por autores de la Universidad UNAN-LEÓN.

Solución:



PRÁCTICAS DE LABORATORIO

IX. PRÁCTICAS DE LABORATORIO

Introducción

Es importante mencionar que a medida que se va desarrollando la asignatura en su parte teórica, también es necesario ir realizando prácticas de laboratorio que ayuden a profundizar mejor los conocimientos. Se diseñaron 7 prácticas en relación con cada tema.

En las primeras dos practicas estudiaremos la introducción al entorno de desarrollo del programa SWI-Prolog-Editor 5.6.48, con el cual se trabajó durante todo el desarrollo de este soporte, se le mostrará al estudiante el manejo básico del compilador, y se realizaran una serie de ejercicios que tienen como objetivo fundamental adaptarse al lenguaje y los distintos comandos básicos. En los siguientes laboratorios el estudiante ejercitará con respecto a cada uno de los conceptos estudiados en cada unidad.

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.

Objetivos de las prácticas de laboratorio

- Complementar los conocimientos adquiridos en las clases teóricas de la asignatura.
- Familiarizar al estudiante en el manejo del entorno de desarrollo SWI-Prolog-Editor 5.6.48.

Duración del laboratorio

Los laboratorios están diseñados con una frecuencia de 2 horas semanales. Al inicio de cada práctica el profesor responsable del laboratorio explicará el contenido de la práctica y deberá estar presente para aclarar dudas presentadas por los estudiantes.

Herramientas Software necesarias

- SWI-Prolog-Editor 5.6.48

PRÁCTICA 1: Descripción e instalación del entorno de desarrollo de SWI-Prolog-Editor

Objetivos

Con la realización de esta práctica, el alumno:

- Aprenderá a instalar el interprete SWI-Prolog-Editor 5.6.48
- Conocerá el manejo básico del entorno de desarrollo SWI-Prolog-Editor 5.6.48.

Tiempo de realización de la práctica

1 sesión de laboratorio (2 horas)

Herramientas Software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

1. Introducción

Prolog es un lenguaje de programación seminterpretado. Su funcionamiento es muy similar a Java. El código fuente se compila a un código de byte el cuál se interpreta en una máquina virtual denominada Warren Abstract Machine (comúnmente denominada WAM).

Por eso, un entorno de desarrollo Prolog se compone de:

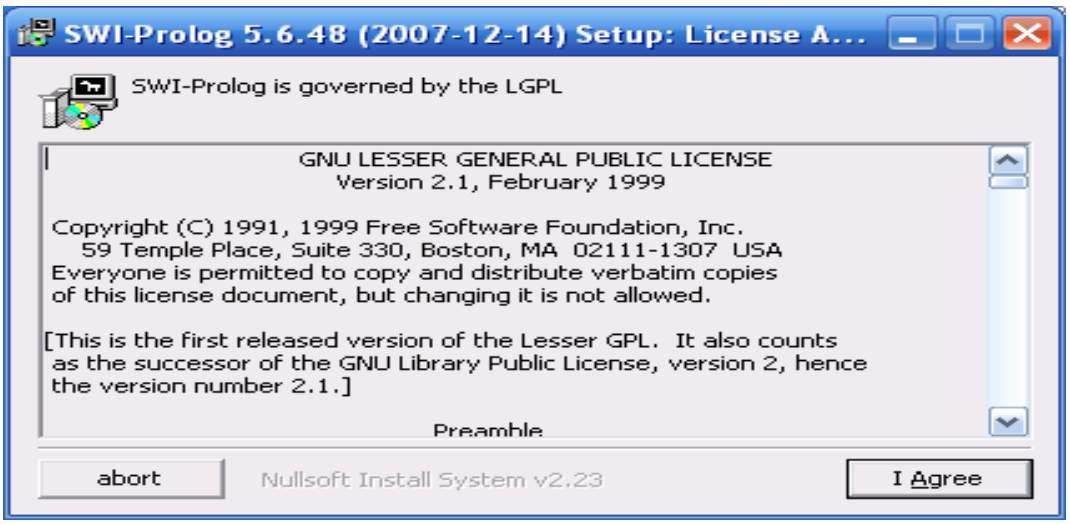
1) Un compilador. Transforma el código fuente en código de byte. A diferencia de Java, no existe un Standard al respecto. Por eso, el código de byte generado por un entorno de desarrollo no tiene por que funcionar en el intérprete de otro entorno.
2) Un intérprete. Ejecuta el código de byte.
3) Un shell o top-level. Se trata de una utilidad que permite probar los programas, depurarlos, etc. Su funcionamiento es similar a los interfaces de línea de comando de los sistemas operativos
4) Una biblioteca de utilidades. Estas bibliotecas son, en general, muy amplias. Muchos entornos incluyen (afortunadamente) unas bibliotecas Standard-ISO que permiten funcionalidades básicas como manipular cadenas, entrada/salida, etc.

Generalmente, los entornos de desarrollo ofrecen extensiones al lenguaje como pueden ser la programación con restricciones, concurrente, orientada a objetos, etc.

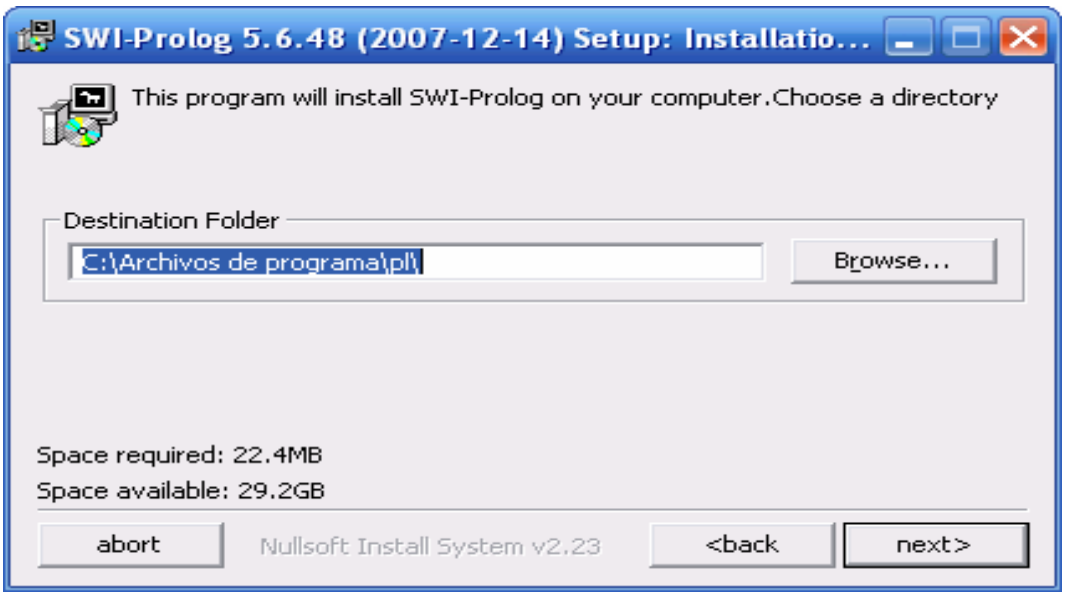
En esta primera práctica se van a presentar los aspectos fundamentales de Prolog, sin entrar en detalle en ninguno de ellos. El objetivo es que tengas un primer contacto con Prolog, con su entorno y que realices los primeros programas.

2. Instalación de SWI-Prolog-Editor 5.6.48

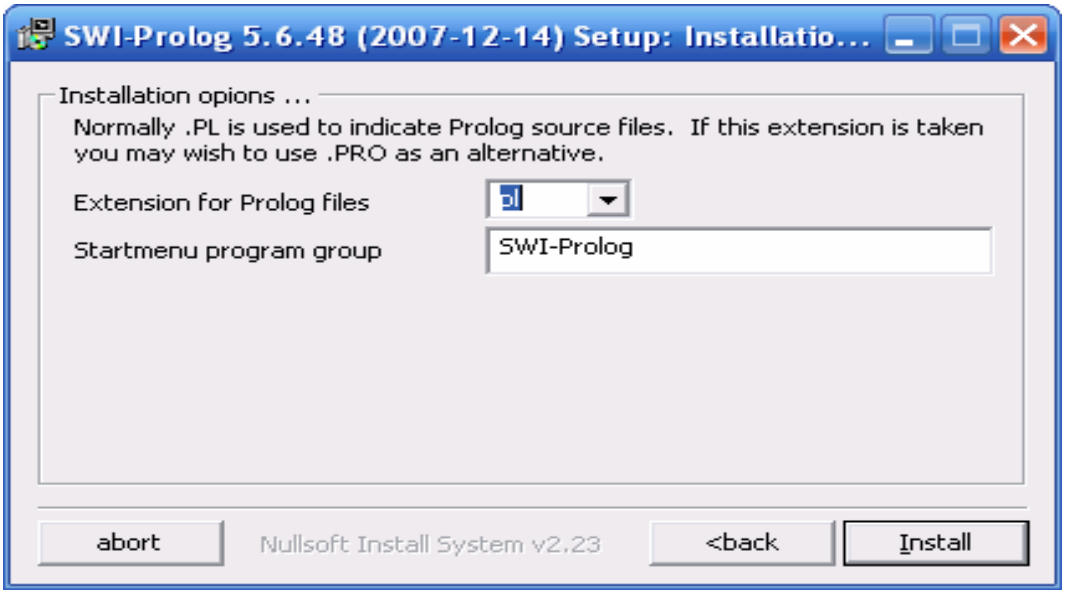
1. Ejecutamos doble clic sobre el Setup.



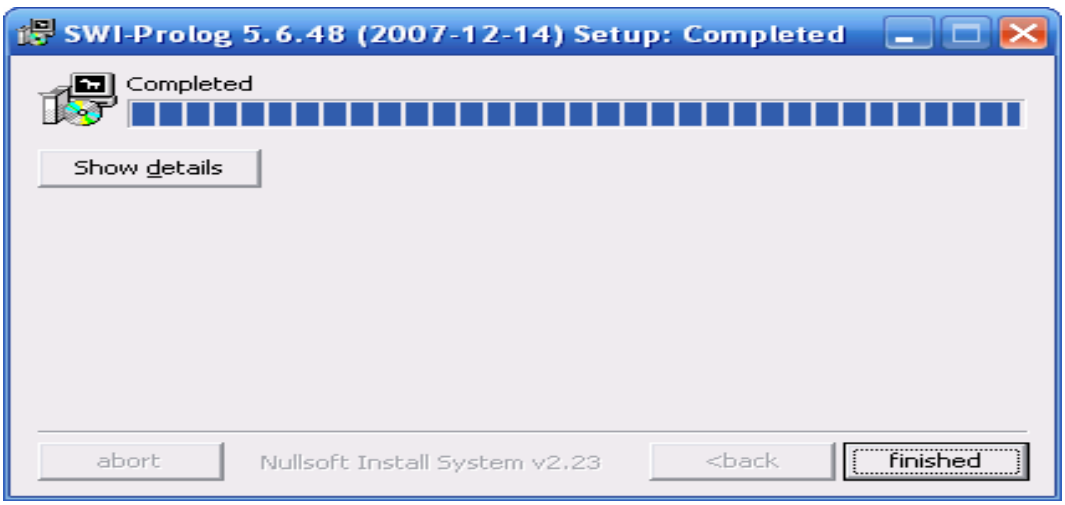
2. El siguiente paso es dar clic en la opción “I Agree”.



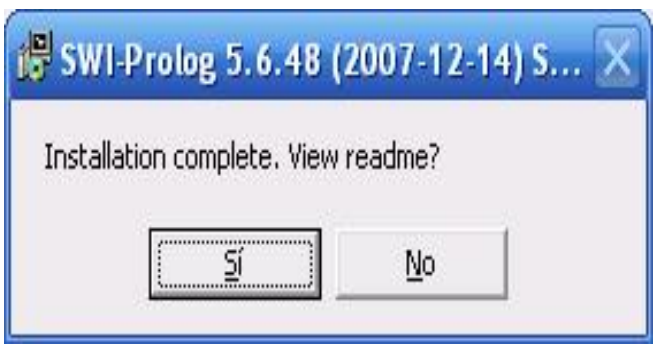
3) Aparecerá una ventana donde tenemos que presionar el botón “ next >”.



4) Luego presionamos la opción "Install". Aparecerá la siguiente ventana, en la cual es necesario dar clic en el botón "finished".

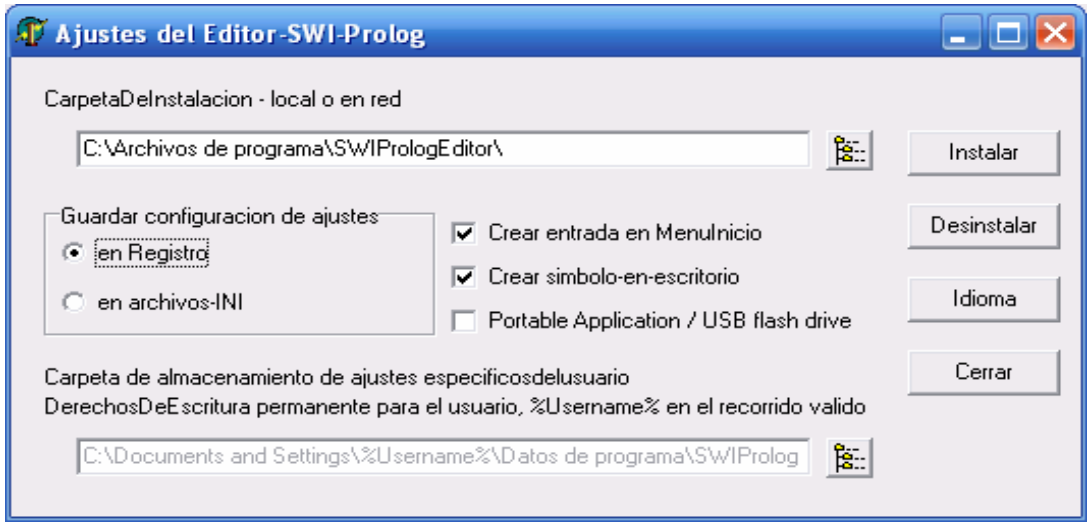


5) Aparecerá una ventana indicando que la instalación ha sido completada.

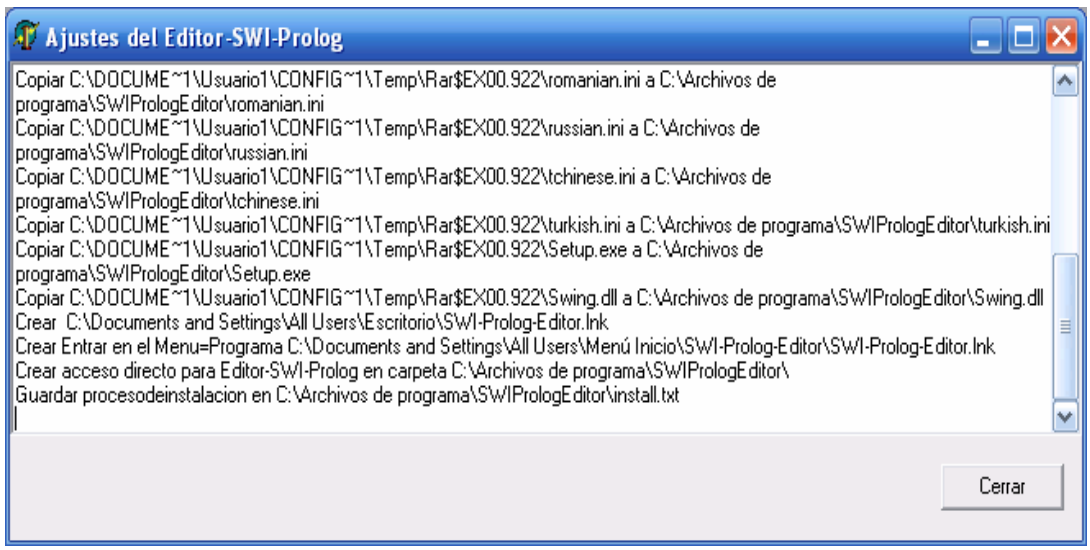


El siguiente paso es instalar el editor, el cual es necesario editar la base de hechos y consultar los archivos .pl.

6) Ejecutar el instalador del SWI-Prolog-Editor 5.6.48



7) Dar clic en el botón Instalar. Con este paso se crea un acceso directo del Editor en el Escritorio de Windows.



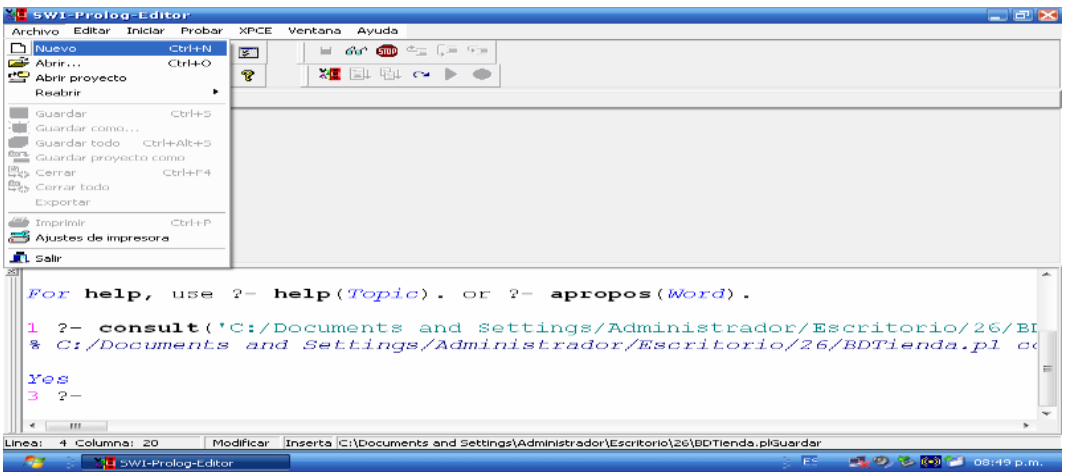
8) Aparecerá una ventana mostrando la configuración del SWI- Prolog-Editor en la cual será necesario dar clic en el botón “Cerrar”.

Como empezar a trabajar con el SWI- Prolog-Editor 5.6.48

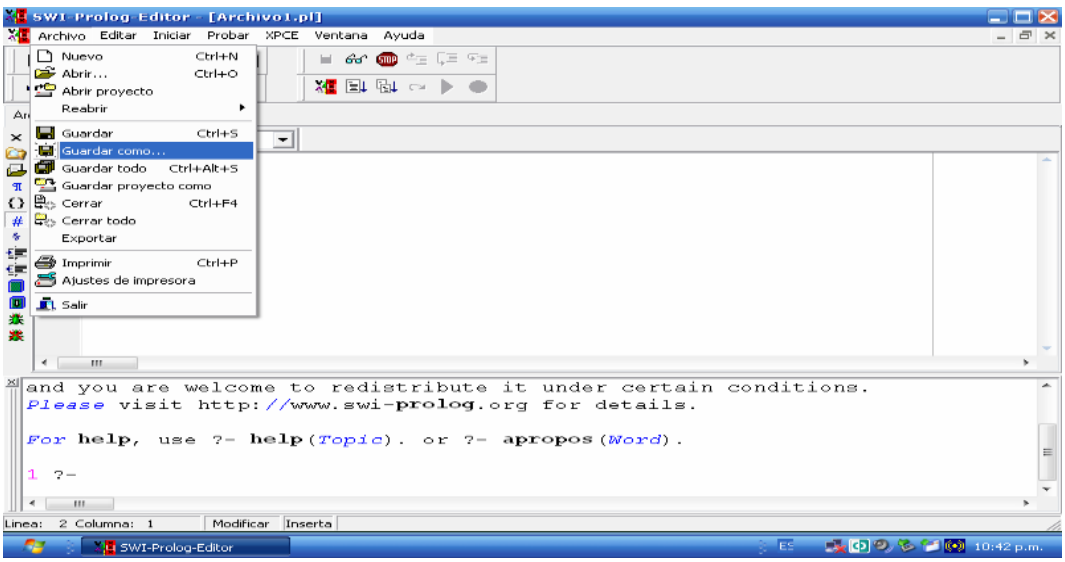
3. Descripción del Entorno de Desarrollo

Una vez iniciado Windows para ejecutar el Editor de Prolog, basta con dar doble click sobre el icono correspondiente (SWI- Prolog-Editor.exe).

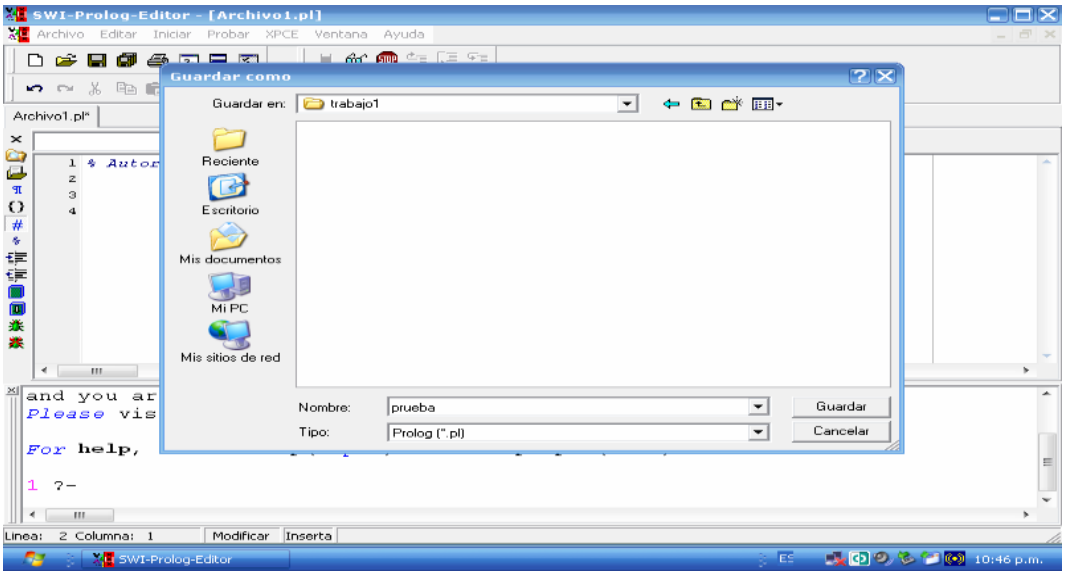
Aparecerá la siguiente ventana en la cual debemos dar click en menú "Archivo" y seleccionar la opción "Nuevo" para crear un nuevo archivo ".pl".



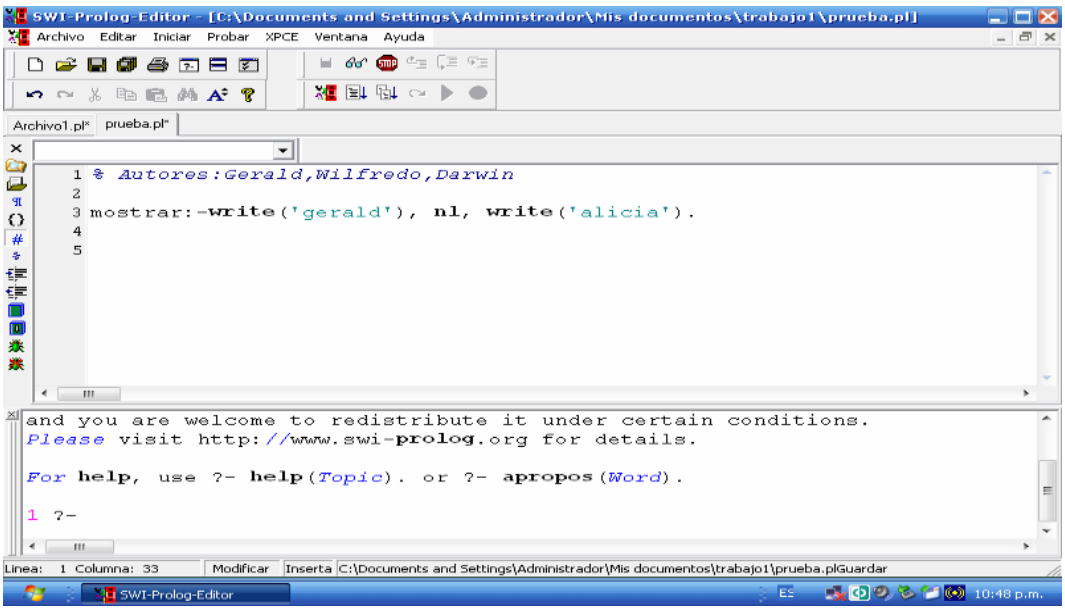
El siguiente paso es guardarlo, dando click en la opción "Guardar Como" del menú Archivo.



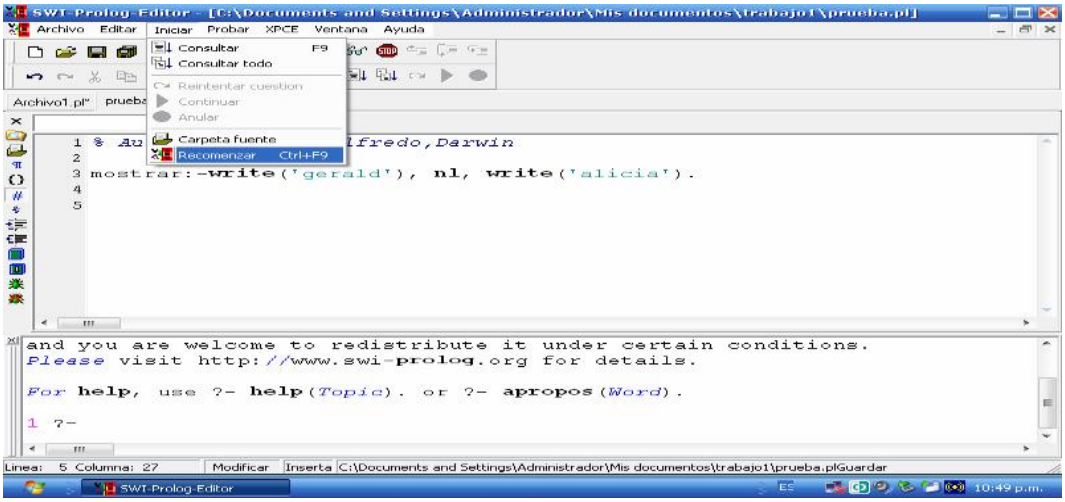
Aparecerá una ventana como la siguiente en la cual se debe crear la carpeta de trabajo donde se guardarán los programas .pl.



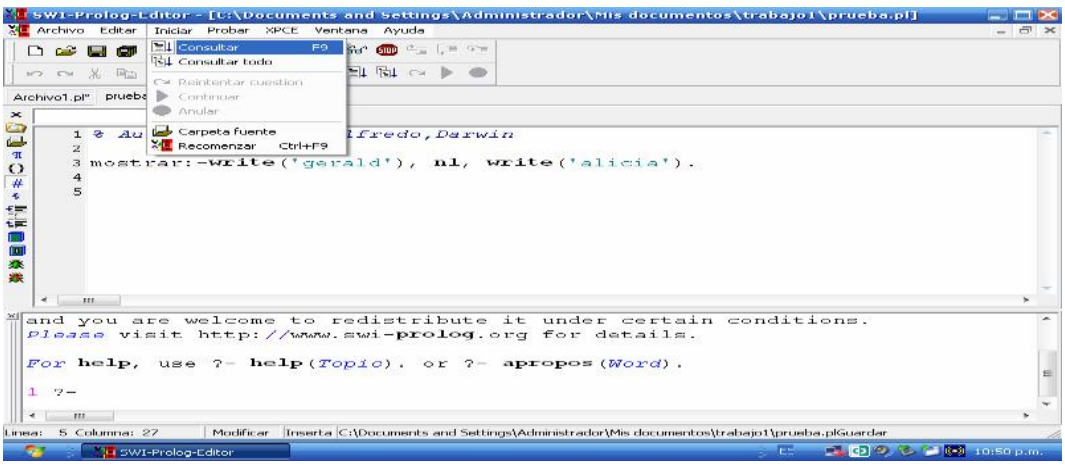
Ahora ya podemos empezar a editar los archivos .pl en el área de trabajo. Como ejemplo escriba lo siguiente: mostrar:- write ('gerald'), nl, write ('alicia').



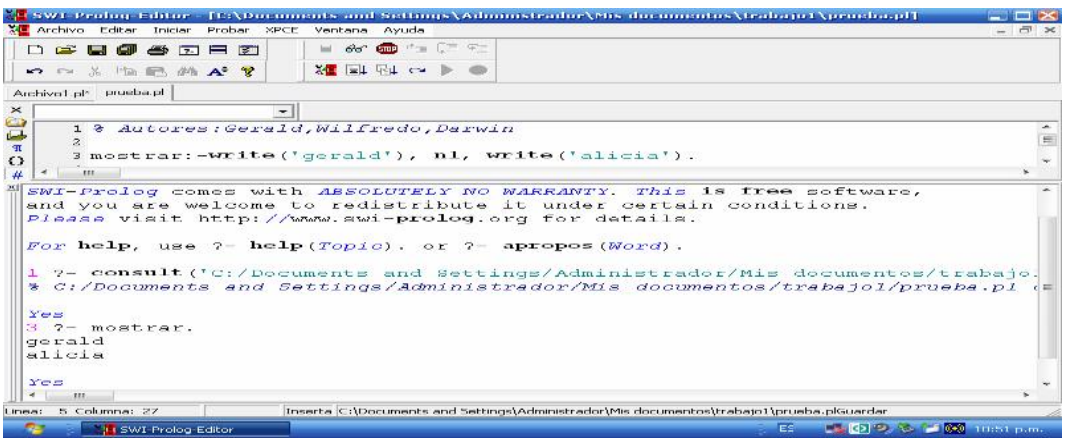
Después debemos de compilarlo para consultar la base de hecho. Primero se hace una precompilación en el Menú "Iniciar" y seleccionamos la opción "Recomenzar".



Para compilar el archivo ".pl" seleccionamos la opción "Consultar" del menú "Iniciar".



Por último consultamos la base de hecho en el prompt de Prolog.



PRÁCTICA 2: Introducción al lenguaje Prolog

Objetivo

Con la realización de esta práctica, el alumno:

- Desarrollará ejercicios haciendo uso de predicados predefinidos.

Tiempo de realización de la práctica

1 sesión de laboratorio (2 horas)

Herramientas software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

Prolog es un lenguaje pensado para la computación simbólica, no numérica. Se adapta muy bien a problemas que involucran hechos y relaciones entre ellos, estas características han convertido a Prolog en un lenguaje poderoso para aplicaciones de Inteligencia Artificial y programación no numérica en general.

La forma más sencilla de especificar una relación en Prolog es mediante un conjunto de hechos, cada hecho consta del nombre de la relación (que comienza con una letra minúscula), seguido por una lista de argumentos separados por coma y encerrados entre paréntesis. Cada hecho finaliza con un punto.

Las reglas especifican cosas que son verdaderas si alguna condición es satisfecha. El símbolo:**-** (que se lee “if” o “si”) delimita dos partes dentro de una regla, una izquierda y otra derecha. Así, en una regla podemos diferenciar: una parte de condición (la parte derecha de la regla), una parte de conclusión (la parte izquierda de la regla). La parte de la conclusión también es llamada la cabeza de una cláusula y la parte de condición el cuerpo de la cláusula.

Desarrollo de la práctica

1) Definir la relación **sumar _ números(N, Y)** que permita verificar si Y es la suma de los N primeros números naturales.

Ejemplo:

```
?-sumar_numero(6,Y).
Y=21
Yes
```

2) Definir la relación **división (X, Y, Z)** que permita verificar si Z es el resultado de la división de X e Y.

Ejemplo:

?-divide (100,25,Z).
Z=4

3) Definir la relación **fibonacci(N, X)** que se verifique si X es el N–ésimo término de la sucesión de Fibonacci.

Nota: La sucesión de Fibonacci es 0, 1, 1, 2, 3, 5, 8, 13, 21,. . . en la que cada término, salvo los dos primeros, es la suma de los dos anteriores.

Ejemplo:

?-fibonacci(6,X).
X=8

4) Definir la relación **descomponer(N, X, Y)** que permita verificar si X e Y son dos cubos cuya suma es N y además X es menor o igual que Y.

Ejemplo:

?- descomponer (1008, X,Y).
X = 8,
Y = 1000
Yes

Nota: para la realización de este ejercicio debes definir la relación **es _ cubo**(N) que permita verificar si N es el cubo de un entero.

Ejemplo:

?- es_cubo(1000).
Yes
?- es_cubo(1200).
No

5) Escribir un programa en Prolog que permita imprimir el cubo de un número dado por el usuario.

Ejemplo:

Escriba un número 10.
El cubo del número es 1000
Yes

PRÁCTICA 3: Entrada y salida en Prolog

Objetivo

Con la realización de esta práctica, el alumno:

- Ejercitara los predicados de entrada y salida en la elaboración de una serie de ejercicios para demostrar su utilidad.

Tiempo de realización de la práctica

1 sesión de laboratorio (2 horas)

Herramientas software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

Como ya se ha comentado anteriormente, una clase importante de predicados interesantes debido a sus efectos colaterales son aquellos encargados de la gestión de las interfaces de entrada y salida de datos. En efecto, esta labor solo puede entenderse desde el punto de vista del sistema.

El predicado fundamental de entrada en Prolog es **read(X)**, el cual lee un término del flujo actual de entrada. Este término una vez leído se unifica con X y read es cierto. En cualquier caso, la instanciación de X a un valor exterior al programa se realiza fuera del modelo lógico considerado, puesto que a cada llamada de read(X) el resultado es cierto con un valor eventualmente distinto de la variable. El predicado fundamental de salida en Prolog es **write(X)**, el cual escribe el término X en el flujo de salida actual definido por el sistema operativo subyacente.

Desarrollo de la práctica

1) Escriba un programa en Prolog que permita encontrar el semiperimetro de un triangulo sabiendo que la ecuación matemática que te permite calcularlo es: S=(a+b+c)/2 donde a, b, c son los tres lados del triangulo. Los valores serán introducidos por el usuario por teclado.

Ejemplo:

```
?-semiperimetro.
Introducir los lados del triangulo
|:10.
|:20.
|:30.
El semiperimetro del triangulo es: 30
```

Yes

2) Escriba un programa en Prolog que solicite un número al usuario e imprima su correspondiente cuadrado.

Ejemplo:

?- cuadrados.
¿Escriba un número?
|: 0.
[0, ^2=, 0]
Yes

?- cuadrados.
¿Escriba un número?
|: 10.
[10, ^2=, 100]
Yes

3) Escriba un programa en Prolog que permita introducir a un usuario la cantidad de elementos de una lista, luego imprima ese número en forma de lista y también saque por pantalla esa lista en forma de *'. Al finalizar el programa debe permitir hacer una búsqueda de cualquier elemento existente.

Ejemplo:

?- lista.
Cuantos elementos? 3.
|: 1.
|: 2.
|: 3.
[1][2][3]
*
**
***
Elemento a buscar en la lista: 2.
Si esta
Yes

4) Desarrollar un programa en Prolog que permita pedirle a un usuario que le adivine un número generado aleatoriamente por el. El programa debe de especificar rango de la ubicación del número para dirigir al usuario.

Ejemplo:

?- adivina.
*******ADIVINA UN NÚMERO******

El numero esta entre 0 y 99: 40.
No has llegado. Prueba de nuevo
El numero esta entre 40 y 99: 80.
Te has pasado. Prueba de nuevo
El numero esta entre 40 y 80: 60.
No has llegado. Prueba de nuevo
El numero esta entre 60 y 80: 75.
Te has pasado. Prueba de nuevo
El numero esta entre 60 y 75: 73.
Te has pasado. Prueba de nuevo
El numero esta entre 60 y 73: 66.
No has llegado. Prueba de nuevo
El numero esta entre 66 y 73: 74.
Te has salido de rango. Prueba de nuevo
El numero esta entre 66 y 73: 70.
No has llegado. Prueba de nuevo
El numero esta entre 70 y 73: 72.
¡FELICIDADES! , has acertado
Yes

PRÁCTICA 4: Listas y operaciones en Prolog

Objetivo

Con la realización de esta práctica, el alumno:

- Hará uso de los conocimientos teóricos adquiridos en clase, para implementar el funcionamiento básico de las operaciones realizadas con listas.

Tiempo de realización de la práctica

1 sesión de laboratorio (2 horas)

Herramientas software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

Como hemos dicho anteriormente, una lista es una estructura de datos simple muy usada en la programación no numérica, la cual la podemos definir como una secuencia de elementos tales como: 1, 2, 3, 4, 5. Cuya sintaxis en el lenguaje Prolog consiste en englobarlos entre corchetes.

Ejemplo

[1, 2, 3, 4,5]

La representación interna de la secuencia de estos elementos en el lenguaje es con árboles binarios, donde la rama de la izquierda es el primer elemento o cabeza de la lista y la rama de la derecha es el resto o cola de la lista. De la rama que contiene el resto de la lista también se distinguen dos ramas: la cabeza y la cola. Y así sucesivamente hasta que la rama de la derecha contenga la lista vacía (representado por [ ]).Todo esto separado por el símbolo **|.**

Ejemplo:

[1|2, 3, 4,5]

Desarrollo de la práctica

1) Definir la relación **reducir (L, L1)** que permita verificar si L1 es la lista obtenida sustituyendo cada sucesión de un elemento de L por dicho numero.

Ejemplo:

?- reducir([a,a,a,b,b,b,c,c,c,d,d,d,f,f],L1).

L1 = [a, b, c, d, f]
Yes

Ejercicio 1

2) Definir la relación **multiplicar (L, N, L1)** que permita verificar si L1 es la lista obtenida multiplicando cada elemento L por N.

Ejemplo:

?-multiplicar([2,6,9,10],5,L1).
L1=[10,30,45,50]
Yes

3) Definir la relación **conteo_resto(X,L1,N,L2)** que permita verificar si N es el número de veces que aparece X en la cabeza de la lista L1 y L2 es el resto de la lista L1 cuando se le quite la sucesión de elementos X de su cabeza.

Ejemplo:

?- conteo_resto(a,[a,a,a,a,b,b,b,c,d,f,f],N,L2).
N = 4,
L2 = [b, b, b, c, d, f, f]
Yes

4) Definir el predicado **reproducir(N, L, L1)** que permita obtener la lista L1 como N reproducciones de la lista L.

Ejemplo:

?- reproducir(2,[a,b,c],L1).
L1 = [a, b, c, a, b, c]
Yes
?- reproducir(0,[a,b,c],L1).
L1=[ ]
Yes

5) Definir el predicado **media (L, M)** que permita verificar la media de una lista dada.

Ejemplo:

?- media([10,20,30,40],M).
M= 25

6) Definir el predicado **lista_impar (L, L1)** que permita verificar si L1 es la lista resultante que contiene solo los elementos que ocupan una posición impar en la lista L.

Ejemplo:

?- lista _ impar([0,1,2,3,4,5,6,7,8,9,10],L1).
L1 = [0, 2, 4, 6, 8, 10];
No

7) Definir la relación **ganancia (L, G)** que permita verificar si G es la ganancia obtenida, según las ofertas de la lista L.

Ejemplo:

?- ganancia([a, e],G).
G = 90
?- ganancia ([a,b],G).
G = 70

Nota: Dada una subasta en la cual se hacen distintas ofertas. Y cada oferta incluye un lote de productos y un precio por dicho lote. En donde las ofertas realizadas se representan mediante la siguiente relación: (Redactar Mejor)

oferta(a, [1, 2,3],40).
oferta(b,[1,2,3],30).
oferta(c,[4],30).
oferta(d,[2,4],30).
oferta(e,[1,2],50).

PRÁCTICA 5: Estructuras de control

Objetivo

Con la realización de esta práctica, el alumno:

- Pondrá en uso los predicados predefinidos corte, fallo y la negación.

Tiempo de realización de la práctica

1 sesión de laboratorio (2 horas)

Herramientas software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

En el lenguaje Prolog existen predicados que se describen como aquellos que ya están definidos y que no necesitamos volver a especificar mediante cláusulas.

A continuación se da una lista de aquellos predicados predefinidos que son muy utilizados, y clasificados según su función.

var: El objetivo var(X) se cumple si X es una variable no instanciada.
novar: El objetivo novar(X) se cumple si X es una variable instanciada.
atom: El objetivo atom(X) se cumple si X representa un átomo PROLOG.
integer: El objetivo integer(X) se cumple si X representa un número entero.
atomic: El objetivo atomic(X) se cumple si X representa un entero o un átomo.

Estos predicados pueden ser controlados por otros predicados. Entre estos cabe mencionar los siguientes:

!: El símbolo de “corte” es un predicado predefinido que fuerza al sistema Prolog a mantener ciertas elecciones que ha realizado.
fail: Este objetivo siempre fracasa.
not: Suponiendo que X está instanciada a un término que pueda interpretarse como un objetivo. El objetivo not(X) se cumple si fracasa el intento de satisfacer X. El objetivo not(X) fracasa si el intento de satisfacer X tiene éxito.
En esta práctica se realizarán varios ejercicios haciendo uso de esto predicado.

Desarrollo de la práctica

1) Definir la relación **únicos (L, L1)** que permita verificar si L1 es la lista de los elementos que ocurren solamente una vez en la lista L.

Ejemplo:

```
?- únicos([3,6,8,3],L1).
   L1 = [6, 8]
?- únicos([3,6,6,8,8,3],L1).
   L1 = []
```

2) Definir la relación **división (Dividendo, Divisor, Res)** que permita verificar si Cociente es el resultado de la división entera entre Dividendo y Divisor.

Nota: Para la realización de este predicado se debe definir la relación natural (Numero) que permite verificar si Numero es un número perteneciente a los Naturales.

Ejemplo:

```
?- división(6,2,Res).
   Res = 3
?- división(100,20,Res).
   Res = 5
```

3) Definir la relación **separar _ números (L1, L2, L3)** que permita separar la lista de números L1 en dos listas en donde L2 estará formada por los números positivos y L3 por los números negativos.

Ejemplo:

```
?- separar([4,0,-6,9,-5,8],L2,L3).
L2 = [4, 9, 8],
L3 = [0, -6, -5]
```

4) Definir una relación **conjunto (L, Conj)** que permita verificar si Conj es el conjunto correspondiente a la lista L.

Nota: Un Conjunto es aquel que contiene los mismos elementos que la lista L en un mismo orden, pero cuando L tenga elementos repetidos solo se debe incluir en Conj la última aparición de cada elemento).

Ejemplo:

```
?- conjunto([a,b,a,d,b,c],Conj).
   Conj = [a, d, b, c]
   Yes
```

5) Consideremos la función siguiente definida sobre los números naturales:

$$f(X)=\begin{cases} 3X+1, \text{ si } X \text{ es impar;} \\ X/2, \text{ si } X \text{ es par} \end{cases}$$

Se pide:

Definir la relación **sucesión(X, L)** que permita verificar si L es la lista de los elementos **X,** $f^{x}, f(f(X)),\ldots, f^{n}(X)$ *tal que* $f^{n}(X) = 1$.

Ejemplo:

?- sucesión(5,L).
L = [5, 16, 8, 4, 2,1]

En donde L se llama la sucesión generada por N.

PRÁCTICA 6: Árboles

Objetivos

Con la realización de esta práctica, el alumno:

- Dará solución a diferentes ejercicios utilizando la metodología de los árboles.
- Aplicará de forma práctica los conocimientos adquiridos en esta unidad y en unidades anteriores.

Tiempo de realización de la práctica

2 sesiones de laboratorio (4 horas)

Herramientas software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

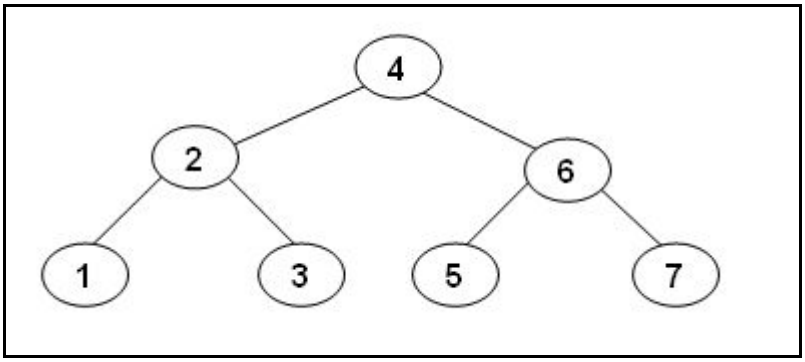
A como se mencionó en la parte teórica de árboles, el uso de estos es muy importante en diferentes aplicaciones, ya que el manejo de estos facilita en gran manera el almacenamiento, la búsqueda y el borrado de diferentes tipos datos.

Los árboles al ser una estructura de datos constituyen uno de los pilares básicos de la programación. Partiendo de los conocimientos adquiridos en los diferentes temas estudiados en este documento incluyendo el tema de árboles, se pretende que los alumnos desarrollen habilidades de resolución de problemas tales como el análisis, la comprensión y codificación de una serie de ejercicios propuestos en esta práctica.

Desarrollo de la práctica

1) Definir la relación **máximo (T, M)** que verifique si M es el valor máximo de los nodos del árbol.

El árbol de resolución se presenta a continuación:



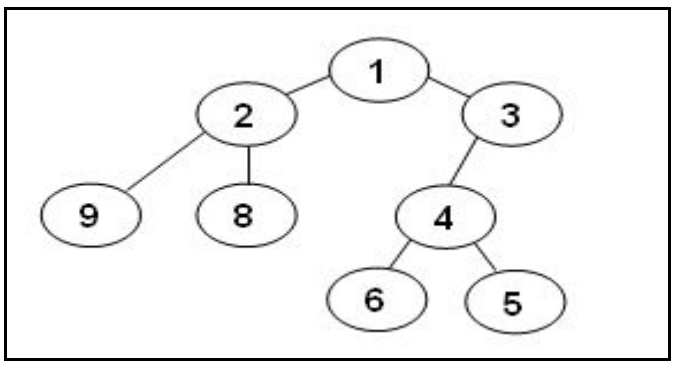
Ejemplo:

?- maximo(vacio,M).
M = 0
?- maximo(arbol(arbol(arbol(vacio, 1, vacio),2,arbol(vacio, 3, vacio)),4,
arbol(arbol(vacio, 5, vacio),6,arbol(vacio, 7, vacio))),M).
M = 7

2) Dada la regla:

arbol(arbol((arbol(vacio,9,vacio),2,arbol(vacio,8,vacio)),1,
arbol(arbol(arbol(vacio,6,vacio),4,arbol(vacio,5,vacio)),3,vacio)).

que representa el árbol siguiente:



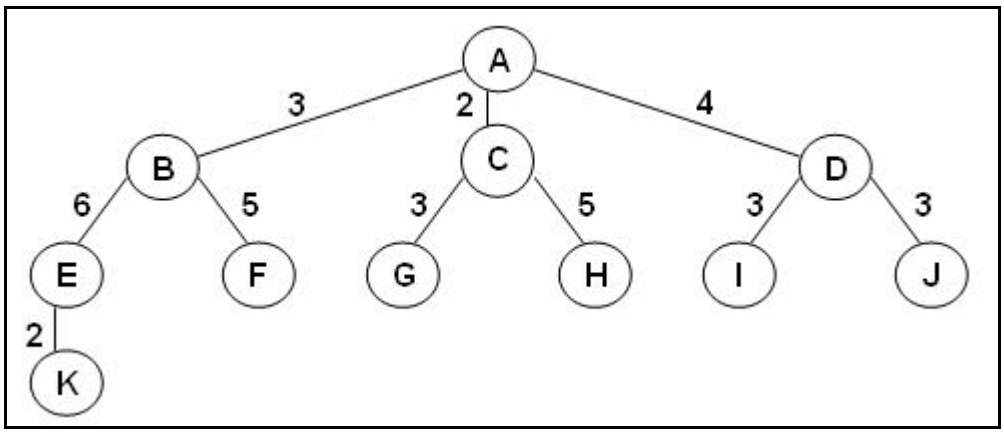
Definir la relación **generación(N, L1, L)** que verifique si L es la lista de nodos de la generación N de la lista de árboles L1, por ejemplo:

?- generacion(1,[arbol(arbol(arbol(vacio,9,vacio),2,arbol(vacio,8,vacio)),1,
arbol(arbol(arbol(vacio,6,vacio),4,arbol(vacio,5,vacio)),3,vacio))],L).

L = [2, 3]

?- generacion(2,[arbol(arbol(arbol(vacio,9,vacio),2,arbol(vacio,8,vacio)),1,
arbol(arbol(arbol(vacio,6,vacio),4,arbol(vacio,5,vacio)),3,vacio))],L).
L = [9, 8, 4]

3) Dado un árbol, cuyas aristas tendrán un costo asociado, desarrolle un predicado que permita obtener un camino valido desde la raíz hasta un nodo hoja y el costo total asociado al mismo. El árbol se representa en la siguiente figura:

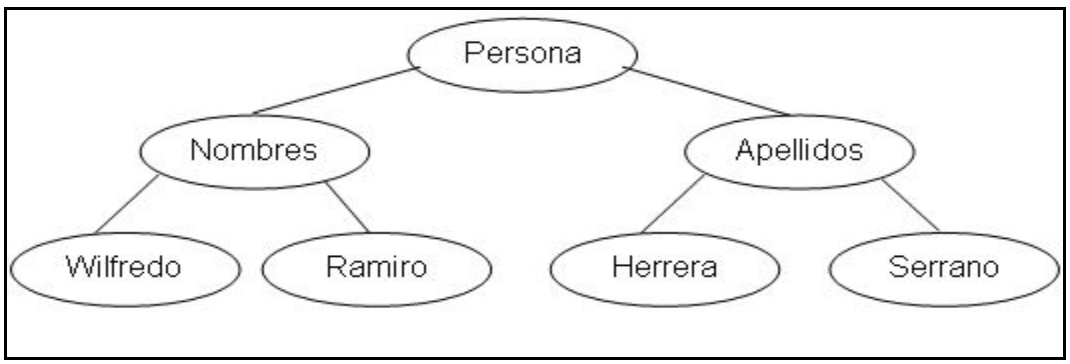


Por ejemplo, el camino desde el nodo raíz 'A' hasta el nodo 'K' tiene un costo de 3+6+2 = 11.

Para representar el árbol se utilizará la siguiente consulta:

ruta_arbol(arbol(a,[arbol(b,[arbol(e,[arbol(k,[],[])],[2]),arbol(f,[],[])],[6,5]),
arbol(c,[arbol(g,[],[]),arbol(h,[],[])],[3,5]),arbol(d,[arbol(i,[],[]),
arbol(j,[],[])],[3,3])],[3,2,4]),L,C).

4) Definir un predicado que arme una lista con todos los elementos del árbol siguiente:



Ejemplo:

```
?- lista_arbol((arbol(Persona,arbol(Nombres,arbol(Wilfredo,vacio,vacio),
arbol(Ramiro,vacio,vacio)),arbol(Apellidos,arbol(Herrera,vacio,vacio),
arbol(Serrano,vacio,vacio)))),L).
L = [Persona,Nombres,Wilfredo,Ramiro,Apellidos,Herrera,Serrano]
```

5. Crear un predicado que determine la cantidad de elementos que tiene el árbol del ejercicio número 2:

Ejemplo:

```
cant_elem_arbol(arbol(1,arbol(2,arbol(9,vacio,vacio),arbol(8,vacio,vacio)),
arbol(3,(arbol(4,arbol(6,vacio,vacio),arbol(5,vacio,vacio))),vacio)),C).
C = 8
```

PRÁCTICA 7: Programación lógica y bases de datos

Objetivo

Con la realización de esta práctica, el alumno:

- Dará solución a diferentes ejercicios utilizando las operaciones fundamentales.

Tiempo de realización de la práctica

1 sesión de laboratorio (2 horas)

Herramientas software necesarias

SWI-Prolog-Editor 5.6.48

Introducción teórica

Como hemos mencionado anteriormente una base de datos es un sistema para archivar información en una computadora cuyo propósito general es mantenerla y hacer que esté disponible cuando se solicite. Las bases de datos son un área de la computación que ha recibido mucha atención debido a sus múltiples aplicaciones. El Algebra relacional es un lenguaje de consulta procedimental y sus diversas operaciones pueden ser desarrolladas en el lenguaje Prolog.

Desarrollo de la práctica

Ejercicio 1:

El conjunto de tiendas de electrodomésticos de la ciudad de chichigalpa desea desarrollar una Base de Dato con las siguientes relaciones:

Tienda (CODTDA, NOMTDA, DIRECCION, TELEFONO).
ELECTRODOMESTICO (CODE, DESCRIPCION, PRECIO).
CLIENTE (CODCLI, NOMCLI, DIRDOMICILIO).
VENTA (CODTDA, CODE, CODCLI, DESCUENTO).

En donde la relación tienda contiene los atributos (nombre de la tienda, dirección, teléfono) que caracterizan la ubicación de la tienda en dicha ciudad, la relación electrodoméstico tiene los atributos código de los electrodomésticos, los diferentes productos que ofertan así como sus correspondientes precios. La relación cliente contiene los datos generales (número de cedula, nombre, dirección domicilio) necesarios de sus respectiva clientela. La relación venta que contiene las llaves primarias de las otras relaciones y también los descuento que se le pueden realizar a un respectivo cliente según la compra que haga.

Se pide formular en álgebra relacional las siguientes consultas:

1) Obtener los nombres y domicilios de los clientes que hayan comprado un electrodoméstico con un descuento mayor del 10%.

2) Obtener las descripciones y precios de aquellos electrodomésticos que han sido vendidos en todas las tiendas.

3) Obtener la descripción de los electrodomésticos vendidos con descuentos del 10% o del 15%.

4) Obtener nombres y teléfonos de las tiendas que no han vendido el electrodoméstico con código 4293-t.

Ejercicio 2:

Considerando la Base de Datos compuesta por las siguientes relaciones:

Profesor (Codp, Nomp, Despacho).
Asistencia (Codp, CodA, CodC).
Asignatura (CodA, Materia).
Clases (CodC, Piso, Bloque).

En donde la relación profesor contiene el código del profesor, el nombre del docente, y el despacho donde se puede ubicar para cualquier información. La relación asignatura contiene las materias que se imparten con sus respectivos códigos, la relación clases tiene la información de cada unos de las aulas donde se imparten las materias y la relación asistencia es donde toas se relacionan.

1) Obtener los profesores que asisten a la clase C1.

2) Obtener los valores de PISO y BLOQUE para las clases a las que asiste el profesor P1.

3) Obtener los valores de CODP para los profesores que asisten a la clase C1 impartiendo la asignatura de física.

4) Obtener los nombres de los profesores que asisten a las clases C2 o C3.

5) Obtener los nombres y el número del despacho de los profesores que asisten a todas las clases del bloque 2.

EJEMPLO DE APLICACIÓN INTELIGENTE EN PROLOG

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 0.El juego lo gana el primero que consigue colocar 3 de sus fichas en línea, ya sea una fila, una columna o una de las diagonales del tablero.

Solución

```
/* Un tablero se encuentra estructurado de la siguiente forma:
* tablero(A,B,C,D,E,F,G,H,I).
* Por ejemplo, tablero(x, 0, x, _, 0, _, 0, _, _) la representación seria:
*
* x | 0 | x
* -----------
*    | 0 |
* -----------
*  0|   |
*
*/

iniciarjuego :- nl, write('BIENVENIDO AL JUEGO DE X,0.'), nl, exp_mov, !, juegos,
!, nl, nl.

exp_mov:- nl, write ('Visualiza la siguiente tabla para poder jugar,'),
nl, write ('para el movimiento que desees hacer:'), nl, nl,
write(' 1 | 2 | 3'), nl, imprime_linea,
write(' 4 | 5 | 6'), nl, imprime_linea,
write (' 7 | 8 | 9'), nl.
/* Muestra los números en el tablero*/

imprime_linea :- write(' -----------'), nl.
juegos:- juego,!, opcion_juego.
opcion_juego:- nl, write ('Desea jugar otra partida (y/n)? '),
entrada_resp (Respuesta),!, proceso(Respuesta).
entrada_resp (Respuesta):- read(Contestacion), verif_resp(Contestacion,
Respuesta), !.
/* Chequea el error y lo pone en su forma normal*/
entrada_resp (Respuesta):- write ('No es una entrada valida teclee y o n: '),
entrada_resp (Respuesta). /* Vuelve a capturar la entrada*/

/* Las entradas permitidas son Si y No: */
verif_resp(y, yes). verif_resp('Y', yes). verif_resp(yes, yes).
verif_resp('YES', yes). verif_resp('Yes', yes).
verif_resp(n, no). verif_resp('N', no). verif_resp(no, no).
verif_resp('NO', no). verif_resp('No', no).
```

```
proceso(no) :- !. /*Fin del programa*/
proceso(yes) :- juegos. /* Continuación del juego*/

juego :- nl, write('con que deseas jugar x o 0? '), entrada_jugador(Jugador), nl,
write('Deseas jugar primero (y/n)? '), entrada_resp(Mov_jugador), !,
jugar(Jugador, Mov_jugador, 9, tablero(_,_,_,_,_,_,_,_,_)).
%Inicializa el tablero con los cuadros vacíos originalmente son nueve

imprimir_tablero(tablero(V1, V2, V3, V4, V5, V6, V7, V8, V9)) :-
write(' '), escribir_Valor(V1), write('|'), escribir_Valor(V2), write('|'),
escribir_Valor(V3), nl, imprime_linea,
write(' '), escribir_Valor(V4), write('|'), escribir_Valor(V5), write('|'),
escribir_Valor(V6), nl, imprime_linea,
write(' '), escribir_Valor(V7), write('|'), escribir_Valor(V8), write('|'),
escribir_Valor(V9), nl, nl.

escribir_Valor(V) :- var(V), !, write(' ').
escribir_Valor(x) :- write(' x ').
escribir_Valor(0) :- write(' 0 ').

entrada_jugador(Jugador) :- read(Jugador), x_or_o(Jugador). /* chequea el error de
entrada del jugador*/
entrada_jugador(Jugador) :- write('No es una respuesta valida teclee  x o 0: '),
entrada_jugador(Jugador). /* vuelve a capturar la entrada */

x_or_o(x). x_or_o(0).

jugar(Jugador, Mov_jugador, Num_apertura, TABLERO) :-
primer_Movimiento(Jugador, Mov_jugador, TABLERO), !,
negacion(Mov_jugador, Nuevo_movjugador),
Apertura is Num_apertura - 1,
seguir_jugando(Jugador, Nuevo_movjugador, Apertura, TABLERO).

primer_Movimiento(Jugador, yes, TABLERO) :- !, movimiento_entrada(TABLERO,
Estado),
realiza_movimiento(m(Estado, Jugador), TABLERO). /* Movimiento del jugador */
primer_Movimiento(Jugador, no, TABLERO) :- opuesto(Jugador, Ordenador),
generar_Mov(Ordenador, TABLERO, Estado),
write('Movimiento del ordenador: '), write(Estado), nl,
realiza_movimiento(m(Estado, Ordenador), TABLERO). /* Movimiento del
Ordenador */

negacion(yes, no). negacion(no, yes).

opuesto(x, 0). opuesto(0, x).
```

```
movimiento_entrada(TABLERO, Estado) :- nl, write('Movimiento del Jugador: '),
read(Estado),
estado_Tablero(Estado), argumentos_Tablero(Estado, TABLERO, Val), var(Val). /*
Realizar otra Apertura */
movimiento_entrada(TABLERO, Estado) :- nl, write('No es valida la entrada.'),
movimiento_entrada(TABLERO, Estado). /* regresar al estado_Tablero */

estado_Tablero(1). estado_Tablero(2). estado_Tablero(3). estado_Tablero(4).
estado_Tablero(5).
estado_Tablero(6). estado_Tablero(7). estado_Tablero(8). estado_Tablero(9).

argumentos_Tablero(1, tablero(Val,_,_,_,_,_,_,_,_), Val) :- !.
argumentos_Tablero(2, tablero(_,Val,_,_,_,_,_,_,_), Val) :- !.
argumentos_Tablero(3, tablero(_,_,Val,_,_,_,_,_,_), Val) :- !.
argumentos_Tablero(4, tablero(_,_,_,Val,_,_,_,_,_), Val) :- !.
argumentos_Tablero(5, tablero(_,_,_,_,Val,_,_,_,_), Val) :- !.
argumentos_Tablero(6, tablero(_,_,_,_,_,Val,_,_,_), Val) :- !.
argumentos_Tablero(7, tablero(_,_,_,_,_,_,Val,_,_), Val) :- !.
argumentos_Tablero(8, tablero(_,_,_,_,_,_,_,Val,_), Val) :- !.
argumentos_Tablero(9, tablero(_,_,_,_,_,_,_,_,Val), Val).

realiza_movimiento(m(Estado, Val), TABLERO) :- argumentos_Tablero(Estado,
TABLERO, Val),
imprimir_tablero(TABLERO).

posicion(1, 1, 2). posicion(2, 1, 6). posicion(3, 1, 10).
posicion(4, 3, 2). posicion(5, 3, 6). posicion(6, 3, 10).
posicion(7, 5, 2). posicion(8, 5, 6). posicion(9, 5, 10).

generar_Mov(Ordenador, TABLERO, Estado) :- division(Pos1, Pos2, Pos3),
argumentos_Tablero(Pos1, TABLERO, Val1), argumentos_Tablero(Pos2,
TABLERO, Val2), argumentos_Tablero(Pos3, TABLERO, Val3),
movimiento_Ganador(Ordenador, Pos1, Pos2, Pos3, Val1, Val2, Val3, Estado), !.
%El ordenador chequea si puede ganar con ese movimiento
generar_Mov(_, TABLERO, Estado) :- division(Pos1, Pos2, Pos3),
argumentos_Tablero(Pos1, TABLERO, Val1), argumentos_Tablero(Pos2,
TABLERO, Val2), argumentos_Tablero(Pos3, TABLERO, Val3),
movimiento_Obligado(Pos1, Pos2, Pos3, Val1, Val2, Val3, Estado), !.
%El ordenador obliga a mover
generar_Mov(Ordenador, TABLERO, Pos3) :- diagonal(Pos1, Pos2, Pos3),
argumentos_Tablero(Pos1, TABLERO, Val1), argumentos_Tablero(Pos2,
TABLERO, Val2), argumentos_Tablero(Pos3, TABLERO, Val3),
proximo(Pos3, NbrA, NbrB),
argumentos_Tablero(NbrA, TABLERO, ValA), argumentos_Tablero(NbrB,
TABLERO, ValB),
movimiento_Especial(Ordenador, Val1, Val2, Val3, ValA, ValB), !.
```

```
%El ordenador intenta conseguir dos de sus movimientos seguidos a lo largo de
una diagonal
%y verifica las casillas ocupadas por el jugador y se coloca en la esquina
generar_Mov(Ordenador, TABLERO, Pos3) :- diagonal(Pos1, Pos2, Pos3),
argumentos_Tablero(Pos1, TABLERO, Val1), argumentos_Tablero(Pos2,
TABLERO, Val2), argumentos_Tablero(Pos3, TABLERO, Val3),
proximo(Pos3, NbrA, NbrB),
argumentos_Tablero(NbrA, TABLERO, ValA), argumentos_Tablero(NbrB,
TABLERO, ValB),
movimiento_Especial2(Ordenador, Val1, Val2, Val3, ValA, ValB), !.
%El ordenador intenta conseguir dos de sus movimientos seguidos verificando el
estado
%del tablero para moverse en una esquina y ver el proximo mas cercano
generar_Mov(Ordenador, TABLERO, Estado) :- division(Pos1, Pos2, Pos3),
argumentos_Tablero(Pos1, TABLERO, Val1), argumentos_Tablero(Pos2,
TABLERO, Val2), argumentos_Tablero(Pos3, TABLERO, Val3),
movimiento_Bueno(Ordenador, Pos1, Pos3, Val1, Val2, Val3, Estado), !.
%Ordenador intenta conseguir 2 seguidos con 3 cuadros desde la apertura
generar_Mov(_, TABLERO, 5) :- argumentos_Tablero(5, TABLERO, Val), var(Val),
!.
%El ordenador toma el centro del tablero si empieza el
generar_Mov(_, TABLERO, Estado) :- esquina_Abierta(TABLERO, Estado), !.
%El ordenador prefiere la posicion de la esquina
generar_Mov(_, TABLERO, Estado) :- estado_Tablero(Estado),
argumentos_Tablero(Estado, TABLERO, Val), var(Val), !. %Toma cualquier cuadro
desde la apertura

division(1,2,3). division(4,5,6). division(7,8,9). division(1,4,7).
division(2,5,8). division(3,6,9). division(9,5,1). division(7,5,3).

diagonal(1,5,9). diagonal(9,5,1). % lista ambos órdenes
diagonal(3,5,7). diagonal(7,5,3).

proximo(1,2,4). proximo(3,2,6).
proximo(7,4,8). proximo(9,6,8).

%Un movimiento ganador se encuentra si Ordenador tiene 2 seguidos y el
% 3 cuadro esta libre
movimiento_Ganador(Ordenador, Pos1,_,_, Val1, Val2, Val3, Pos1) :- var(Val1),
nonvar(Val2), Val2 == Ordenador, nonvar(Val3), Val3 == Ordenador, !.
movimiento_Ganador(Ordenador,_, Pos2,_, Val1, Val2, Val3, Pos2) :- var(Val2),
nonvar(Val1), Val1 == Ordenador, nonvar(Val3), Val3 == Ordenador, !.
movimiento_Ganador(Ordenador,_,_, Pos3, Val1, Val2, Val3, Pos3) :- var(Val3),
nonvar(Val1), Val1 == Ordenador, nonvar(Val2), Val2 == Ordenador, !.

% El Ordenador se obliga a mover si  el Jugador tiene dos movimientos
%seguidos y el  3 cuadro esta abierto
```

```
movimiento_Obligado(_,_, Pos3, Val1, Val2, Val3, Pos3) :- var(Val3),
nonvar(Val1), nonvar(Val2), Val1 == Val2, !.
movimiento_Obligado(_, Pos2,_, Val1, Val2, Val3, Pos2) :- var(Val2),
nonvar(Val1), nonvar(Val3), Val1 == Val3, !.
movimiento_Obligado(Pos1,_,_, Val1, Val2, Val3, Pos1) :- var(Val1),
nonvar(Val2), nonvar(Val3), Val2 == Val3, !.

movimiento_Especial(Ordenador, Val1, Val2, Val3, ValA, ValB) :-
var(Val2), var(Val3), nonvar(Val1), Val1 == Ordenador,
nonvar(ValA), ValA == Ordenador, nonvar(ValB), ValB == Ordenador, !.
movimiento_Especial(Ordenador, Val1, Val2, Val3, ValA, ValB) :-
var(Val1), var(Val3), nonvar(Val2), Val2 == Ordenador,
nonvar(ValA), ValA == Ordenador, nonvar(ValB), ValB == Ordenador, !.

movimiento_Especial2(Ordenador, Val1, Val2, Val3, ValA, ValB) :-
var(Val2), var(Val3), nonvar(Val1), Val1 == Ordenador,
nonvar(ValA), ValA == Ordenador, var(ValB), !.
movimiento_Especial2(Ordenador, Val1, Val2, Val3, ValA, ValB) :-
var(Val1), var(Val3), nonvar(Val2), Val2 == Ordenador,
var(ValA), nonvar(ValB), ValB == Ordenador, !.
movimiento_Especial2(Ordenador, Val1, Val2, Val3, ValA, ValB) :-
var(Val2), var(Val3), nonvar(Val1), Val1 == Ordenador,
var(ValA), nonvar(ValB), ValB == Ordenador, !.
movimiento_Especial2(Ordenador, Val1, Val2, Val3, ValA, ValB) :-
var(Val1), var(Val3), nonvar(Val2), Val2 == Ordenador,
nonvar(ValA), ValA == Ordenador, var(ValB), !.

%movimiento bueno para que el Ordenador consiga 2 cuadros seguidos
%y encuentre un tercero abierto
movimiento_Bueno(Ordenador,_, Pos3, Val1, Val2, Val3, Pos3) :- var(Val2),
var(Val3), nonvar(Val1), Val1 == Ordenador, !.
movimiento_Bueno(Ordenador, Pos1,_, Val1, Val2, Val3, Pos1) :- var(Val1),
var(Val3), nonvar(Val2), Val2 == Ordenador, !.
movimiento_Bueno(Ordenador, Pos1, _, Val1, Val2, Val3, Pos1) :- var(Val1),
var(Val2), nonvar(Val3), Val3 == Ordenador, !.

esquina_Abierta(TABLERO, Estado) :- esquina(Estado),
argumentos_Tablero(Estado, TABLERO, Val), var(Val).

esquina(1). esquina(3). esquina(7). esquina(9).

seguir_jugando(Jugador,_,_, TABLERO) :- gano(Quien, TABLERO), %chequea si
alguien ha ganado
!, perdedor(Quien, Jugador).
seguir_jugando(Jugador, Mov_jugador, Num_apertura, TABLERO) :-
Num_apertura < 3,
conseguir_Mov(Jugador, Mov_jugador, Mover),
```

```
cuadro_Abierto(TABLERO, Posicion),
empate(Mover, Posicion, TABLERO), !,
write('Has empatado,No hay ganador.'), nl.
/*Si Num_apertura es 1 o 2 chequeamos si hay un empate */
seguir_jugando(Jugador, Mov_jugador, Num_apertura, TABLERO) :-
jugar(Jugador, Mov_jugador, Num_apertura, TABLERO).
/* Continuar  jugando si no hay victoria no se encontrado ningún empate*/

gano(Quien, TABLERO) :- division(Pos1, Pos2, Pos3), argumentos_Tablero(Pos1,
TABLERO, Val1),
nonvar(Val1), Quien = Val1, /* Quien es el ganador */
argumentos_Tablero(Pos2, TABLERO, Val2), nonvar(Val2), Val1 == Val2,
argumentos_Tablero(Pos3, TABLERO, Val3), nonvar(Val3), Val1 == Val3.

perdedor(Jugador, Jugador) :- !, write('Has ganado FELICIDADES!'), nl.
perdedor(_,_) :- write('Ha ganado el ordenador!'), nl.

%El jugador se encuentra listo para mover después que el ordenador
%realiza su movimiento
conseguir_Mov(Jugador, yes, Jugador) :- !.
conseguir_Mov(Jugador, no, Ordenador) :- opuesto(Jugador, Ordenador).

cuadro_Abierto(TABLERO, Posicion) :-
findall(Pos, posicion_Abierta(TABLERO, Pos), Posicion).
%Las posiciones estan lista para la apertura del TABLERO

posicion_Abierta(TABLERO, Pos) :- estado_Tablero(Pos),
argumentos_Tablero(Pos, TABLERO, Val), var(Val).

empate(Mover, Posicion, TABLERO) :-
not(verif_Ganador(Mover, Posicion, TABLERO)).

verif_Ganador(Mover, [Pos1, Pos2], TABLERO) :-
test2(Mover, Pos1, Pos2, TABLERO), !.
%prueba si los últimos 2 movimientos pudieran llevar a una victoria
verif_Ganador(Mover, [Pos1, Pos2], TABLERO) :- !,
test2(Mover, Pos2, Pos1, TABLERO).
verif_Ganador(Mover, [Pos], TABLERO) :- test1(Mover, Pos, TABLERO).
%prueba si último movimiento da un empate

test2(Mover, Pos1, Pos2, TABLERO) :- argumentos_Tablero(Pos1, TABLERO,
Mover),
opuesto(Mover, Next), argumentos_Tablero(Pos2, TABLERO, Next),
ganar(TABLERO).
% prueba si el movimiento a Pos1 y luego mueve a Pos2 lleva al ganador
```

```
test1(Mover, Pos, TABLERO) :- argumentos_Tablero(Pos,TABLERO, Mover),
ganar(TABLERO).
%prueba si el movimiento a Pos lleva al ganador
ganar(TABLERO) :- gano(x, TABLERO), !.
ganar(TABLERO) :- gano(0, TABLERO).
```

Salida después de consultar el programa:

```
?- iniciarjuego.
BIENVENIDO AL JUEGO DE X, 0.
Visualiza la siguiente tabla para poder jugar,
para el movimiento que desees hacer:

 1 | 2 | 3
 -----------
 4 | 5 | 6
 -----------
 7 | 8 | 9

Con que deseas jugar x o 0? x.

Deseas jugar primero (y/n)? y.

Movimiento del Jugador: 5.
  |   |
 -----------
  | x |
 -----------
  |   |

Movimiento del ordenador: 1
  0 |   |
 -----------
    | x |
 -----------
    |   |


Movimiento del Jugador: 9.
  0 |   |
 -----------
    | x |
 -----------
    |   | x

Movimiento del ordenador: 3
  0 |   | 0
```

```
-----------
   | x |
-----------
   |   | x
```

Movimiento del Jugador: 2.

```
 0 | x | 0
-----------
   | x |
-----------
   |   | x
```

Movimiento del ordenador: 8

```
 0 | x | 0
-----------
   | x |
-----------
   | 0 | x
```

Movimiento del Jugador: 6.

```
 0 | x | 0
-----------
   | x | x
-----------
   | 0 | x
```

Movimiento del ordenador: 4

```
 0 | x | 0
-----------
 0 | x | x
-----------
   | 0 | x
```

Has empatado, No hay ganador.
Desea jugar otra partida (y/n)? n.
Yes

Enunciado 2:

El problema de la zorra, la oca, el grano y el campesino consistente en lo siguiente:

Suponga que un campesino tiene una zorra, una oca y un saco de granos, y que desea cruzar un río. Para eso tiene un bote en el que puede cruzar el río con uno solo de los otros elementos (zorra, oca o grano). En cada viaje, en cualquiera de los dos sentidos, debe viajar el campesino ya que es el que debe remar. En un inicio están los 4 de un lado del río y el objetivo final es que estén los 4 del otro lado. Hay dos restricciones importantes que son las siguientes.

Cuando el campesino no está, no pueden quedar juntas la zorra y la oca porque la primera se come a la segunda. Tampoco pueden quedar juntos la oca y el grano porque la primera se come al segundo. El objetivo es encontrar la secuencia de viajes que permita realizar el paso del río.

Sugerencia: representar en hechos los movimientos posibles que pueden hacerse con el bote, y en otros hechos los estados prohibidos.

Solución:

```
% estado inicial
inicio (estado (lado(1, Zorras, Ocas, Granos), lado(0, 0, 0, 0), CapBote)):-
write (’cantidad de Zorras: ’), read(Zorras),
write (’cantidad de Ocas : ’), read(Ocas),
write (’cantidad de Granos : ’), read(Granos),
write(’capacidad del Bote: ’), read(CapBote).
% estado final
fin (estado(lado(0, 0, 0, 0), lado(1, _, _, _), _)).
% restrincion(es) de peligro
% no hay peligro solo cuando en ambos lados los animales estan a salvo
sin_problemas (E):-
E = estado (Li, Lf, _),
es_seguro (Li),
es_seguro (Lf).
% no hay ocas y en la orilla hay zorras y granos
es_seguro (L):-
L = lado (_, Zorras, Ocas, Granos),
Ocas = 0, Zorras > 0, Granos > 0.
% no hay ocas, pero en la orilla hay zorras
es_seguro (L):-
L = lado (_, Zorras, Ocas, _),
Ocas = 0, Zorras > 0.
% no hay ocas, pero en la orilla hay granos
es_seguro (L):-
```

```
L = lado(_, _, Ocas, Granos),
Ocas = 0, Granos > 0.
% no hay zorras ni granos, pero en la orilla hay ocas
es_seguro(L):-
L = lado(_, Zorras, Ocas, Granos),
Ocas > 0, Zorras = 0, Granos = 0.
% las ocas y el hombre estan en la misma orilla
es_seguro(L):-
L = lado(Hombre, _, Ocas, _),
Ocas > 0, Hombre = 1.
% no queda ningun elemento en la orilla
es_seguro(L):-
L = lado(_, Zorras, Ocas, Granos),
Ocas = 0, Zorras = 0, Granos = 0.
% Son movimientos posibles cuando...
% viaje el Hombre de un lado a otro
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
Gf1 is Gi1, Gf2 is Gi2,
Of1 is Oi1, Of2 is Oi2,
%M = '.: vija el Hombre al otro lado :.',
%  ' H,'+Oi1+'O -> '
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ei a Ef cuando Ocas = CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
Oi1 =:= (CB-1),
Of1 is Oi1 - (CB - 1),
Of2 is Oi2 + (CB - 1),
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
Gf1 is Gi1, Gf2 is Gi2,
% viaja el Hombre con más ocas del lado 1 al lado 2
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ef a Ei cuando Ocas = CB-1
```

```
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
 Oi2 =:= (CB-1),
Of1 is Oi1 + (CB - 1),
Of2 is Oi2 - (CB - 1),
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
Gf1 is Gi1, Gf2 is Gi2,
%M = '.: viaja el Hombre con ,'+Oi2+'Ocas del lado 2 al lado 1
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ei a Ef cuando Ocas < CB-1
% y la cantidad de Granos y Zorras caben en el espacio vacio
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es menor que la capacidad del bote
Oi1 < (CB-1), Gi1 =< (CB - 1) - Oi1, Zi1 =< (((CB - 1) - Oi1) - Gi1),
% el granjero se lleva todas las Ocas, las Zorras y Granos que quepan en el bote
Of1 is 0, Of2 is Oi2 + Oi1,
Gf1 is 0, Gf2 is Gi2 + Gi1,
Zf1 is 0, Zf2 is Zi2 + Zi1,
Hf1 is Hi2, Hf2 is Hi1,

%M = '.: viaja el Hombre con '+Zi1+'Zorras,'+Oi1+'Ocas,'+Gi1+'Granos
%del lado 1 al lado 2 :.',
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ei a Ef cuando Ocas < CB-1
% y la cantidad de Zorras cabe en el espacio vacio
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es menor que la capacidad del bote
Oi1 < (CB-1), Zi1 =< (CB - 1) - Oi1,
% el granjero se lleva todas las Ocas y las Zorras
```

```
Of1 is 0, Of2 is Oi2 + Oi1,
Zf1 is 0, Zf2 is Zi2 + Zi1,
Hf1 is Hi2, Hf2 is Hi1,
Gf1 is Gi1, Gf2 is Gi2,
%M = '.: viaja el Hombre con '+Zi1+'Zorras y '+Oi1+'Ocas del
% lado 1 al lado 2 :.',
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ei a Ef cuando Ocas < CB-1
% y la cantidad de Granos cabe en el espacio vacio
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es menor que la capacidad del bote
Oi1 < (CB-1), Gi1 =< (CB - 1) - Oi1,
% el granjero se lleva todas las Ocas y las Zorras
Of1 is 0, Of2 is Oi2 + Oi1,
Gf1 is 0, Gf2 is Gi2 + Gi1,
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
% M = '.: viaja el Hombre con '+Oi1+'Ocas y '+Gi1+'Granos del lado
%1 al lado 2 :.',
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ei a Ef cuando Ocas < CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es menor que la capacidad del bote
Oi1 < (CB-1),
% el granjero se lleva todas las Ocas
Of1 is 0,
Of2 is Oi2 + Oi1,
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
Gf1 is Gi1, Gf2 is Gi2,
%viaja el Hombre con '+Oi1+'Ocas del lado 1 al lado 2
sin_problemas(Ef).
% viaje el Hombre con las Ocas de Ef a Ei cuando Ocas < CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
```

```
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es menor que la capacidad del bote
Oi2 < (CB-1),
% el granjero se lleva todas las Ocas
Of2 is 0,
Of1 is Oi2 + Oi1,
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
Gf1 is Gi1, Gf2 is Gi2,
% M = '.: viaja el Hombre con '+Oi2+'Ocas del lado 2 al lado 1
sin_problemas(Ef).
% viaje el Hombre de Ei a Ef cuando Ocas < CB-1 y Zorras > CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% verifico que Ocas < CB-1 y Zorras > CB-1
Oi1 < (CB - 1), Zi1 > (CB - 1),
% muevo las ocas y las zorras que quepan en el bote
Of1 is 0, Of2 is Oi1 + Oi2,
Zf1 is Zi1 - ((CB - 1) - Oi1), Zf2 is Zi2 + ((CB - 1) - Oi1),

Gf1 is Gi1, Gf2 is Gi2,
Hf1 is Hi2, Hf2 is Hi1,
sin_problemas(Ef).
% viaja el Hombre de Ei a Ef cuando Ocas > CB-1 y no hay Ocas ni Zorras
% en la orilla que cargar
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es mayor que la capacidad del bote
Oi1 >= (CB-1), Zi1 =:= 0, Gi1 =:= 0,
% y no hay Zorras ni Granos en la orilla, el Hombre se lleva las Ocas que caben en
el bote
Of1 is Oi1 - (CB-1),
Of2 is Oi2 + (CB-1),
```

```
Hf1 is Hi2, Hf2 is Hi1,
Zf1 is Zi1, Zf2 is Zi2,
Gf1 is Gi1, Gf2 is Gi2,
sin_problemas(Ef).
% viaje el Hombre de Ei a Ef con otro elemento cuando Ocas > CB-1
% y la suma de las Zorras y las Granos es < CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es mayor que la capacidad del bote
Oi1 >= (CB-1), (Zi1 + Gi1) =< (CB-1),
% y las Zorras + Granos < CB-1, el granjero se los lleva
Zf1 is 0, Zf2 is Zi2 + Zi1,
Gf1 is 0, Gf2 is Gi2 + Gi1,
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
% M = '.: viaja el Hombre con '+Zi1+'Zorras y '+Gi1+'Granos del
% lado 1 al lado 2 :.',
 sin_problemas(Ef).
% viaje el Hombre de Ef a Ei con otro elemento cuando Ocas > CB-1
% y la suma de las Zorras y las Granos es < CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Ocas es mayor que la capacidad del bote
Oi2 >= (CB-1), (Zi2 + Gi2) =< (CB-1),
% y las Zorras + Granos < CB-1, el granjero se los lleva
Zf2 is 0, Zf1 is Zi2 + Zi1,
Gf2 is 0, Gf1 is Gi2 + Gi1,
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
% M = '.: viaja el Hombre con '+Zi2+'Zorras y '+Gi2+'Granos del
% lado 2 al lado 1 :.',
sin_problemas(Ef).
% viaje el Hombre con las Zorras cuando Zorras = CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
```

```
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
 Zi1 =:= (CB - 1),
Zf1 is Zi1 - (CB - 1),
Zf2 is Zi2 + (CB - 1),
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
Gf1 is Gi1, Gf2 is Gi2,
%viaja el Hombre con '+Zi1+'Zorras del lado 1 al lado 2
sin_problemas(Ef).
% viaje el Hombre con las Zorras cuando Zorras < CB-1 y caben Granos
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Zorras son menor que la capacidad del bote y caben Granos
Zi1 < (CB - 1), Gi1 =< (CB - 1) - Zi1,
% el granjero se lleva todas las zorras y los Granos que quepan
Zf1 is 0, Zf2 is Zi2 + Zi1,
Gf1 is 0, Gf2 is Gi2 + Gi1,

Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
%  M = '.: viaja el Hombre con '+Zi1+'Zorras y '+Gi1+' del lado 1
% al lado 2 :.',
sin_problemas(Ef).
% viaje el Hombre con las Zorras cuando Zorras < CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
% si las Zorras son menor que la capacidad del bote
Zi1 < (CB - 1),
% el granjero se lleva todas las zorras
Zf1 is 0,
Zf2 is Zi2 + Zi1,
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
Gf1 is Gi1, Gf2 is Gi2,
%  viaja el Hombre con '+Zi1+'Zorras del lado 1 al lado 2
```

```
sin_problemas(Ef).
% viaje el Hombre con los Granos cuando Granos = CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
Gi1 =:= (CB - 1),
Gf1 is Gi1 - (CB - 1),
Gf2 is Gi2 + (CB - 1),
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
Zf1 is Zi1, Zf2 is Zi2,
%viaja el Hombre con '+Gi1+'Granos del lado 1 al lado 2
sin_problemas(Ef).
% viaje el Hombre con los Granos de Ei a Ef cuando Granos < CB-1
% y caben Zorras
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
Gi1 < (CB - 1),  Zi1 =< (CB - 1) - Gi1,
Gf1 is 0, Gf2 is Gi2 + Gi1,
Zf1 is 0, Zf2 is Zi2 + Zi1,
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
% M = '.: viaja el Hombre con '+Zi1+'Zorras y '+Gi1+'Granos del
% lado 1 al lado 2 :.',
sin_problemas(Ef).
% viaje el Hombre con los Granos de Ei a Ef cuando Granos < CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
Gi1 < (CB - 1),
Gf1 is 0,
Gf2 is Gi2 + Gi1,
Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
```

```
Zf1 is Zi1, Zf2 is Zi2,
% M = '.: viaja el Hombre con '+Gi1+'Granos del lado 1 al lado 2
sin_problemas(Ef).
% viaje el Hombre con los Granos de Ei a  Ef cuando Granos > CB-1
ir_desde(Ei, Ef):-
Ei = estado(Li1, Li2, CB),
Li1 = lado(Hi1, Zi1, Oi1, Gi1),
Li2 = lado(Hi2, Zi2, Oi2, Gi2),
Ef = estado(Lf1, Lf2, CB),
Lf1 = lado(Hf1, Zf1, Of1, Gf1),
Lf2 = lado(Hf2, Zf2, Of2, Gf2),
Gi1 > (CB - 1),
Gf1 is Gi1 - (CB - 1),
Gf2 is Gi2 + (CB - 1),Hf1 is Hi2, Hf2 is Hi1,
Of1 is Oi1, Of2 is Oi2,
Zf1 is Zi1, Zf2 is Zi2,
% Gt is (CB - 1),

%  viaja el Hombre con '+Gt+'Granos del lado 1 al lado 2

sin_problemas(Ef).
% #endregion Datos del problema
mostrar([E]):-
E = estado(L1, L2, _),
L1 = lado(H1, Z1, O1, G1),
L2 = lado(H2, Z2, O2, G2),
write(H1),write('\t'),write(Z1),write('\t'),write(O1),write('\t'),write(G1),
write('   ----   '),
write(H2),write('\t'),write(Z2),write('\t'),write(O2),write('\t'),write(G2),nl.
%write('.: incialmente estan todos en el lado 1 :.'),nl.
mostrar([E|List]):-
mostrar(List),
E = estado(L1, L2, _),
L1 = lado(H1, Z1, O1, G1),
L2 = lado(H2, Z2, O2, G2),
write(H1),write('\t'),write(Z1),write('\t'),write(O1),write('\t'),write(G1),
write('   ----   '),
write(H2),write('\t'),write(Z2),write('\t'),write(O2),write('\t'),write(G2),nl.
% resolver(E, C, R) realiza una busqueda en profundidad donde :
% E: Estado de partida
% C: Camino recorrido (en reversa)
% R: Camino total hasta la solucion (en reversa)
resolver(E, C, C) :- fin(E).
resolver(E, C, R) :- ir_desde(E, E1), not(member(E1, C)),
resolver(E1, [E1|C], R).
solucion:-
nl,
```

```
write(' Problema de la Zorra, la Oca, el Grano y el Hombre'),
nl,nl,
write('- Entrada de datos...'),nl,
inicio(Ei),
Ei = estado(Li, Lf, CB),
Ef = estado(Lf, Li ,CB),
fin(Ef),
E = Ei,
resolver(E, [E], R),nl,
write('- Solucion propuesta'),nl,
write('\tLado 1'),write('\t\t\t'),write('\tLado 2'),nl,
write('H\tZ\tO\tG'),
write('          '),
write('H\tZ\tO\tG'),nl,
mostrar(R),!.
```

Después de correrlo la salida es:

```
?- solución.
 Problema de la Zorra, la Oca, el Grano y el Hombre

Entrada de datos
cantidad de Zorras : 1.
cantidad de Ocas : 1.
cantidad de Granos : 1.
capacidad del Bote : 3.

Solución propuesta

        Lado 1                          Lado 2
H       Z       O       G       H       Z       O       G
1       1       1       1  ----  0       0       0       0
0       0       0       1  ----  1       1       1       0
1       0       1       1  ----  0       1       0       0
0       0       0       0  ----  1       1       1       1

Yes
```

X. CONCLUSIÓN

El objetivo principal de esta tesis fue desarrollar un soporte para el componente curricular de Inteligencia Artificial y Sistemas Expertos basado en el lenguaje de programación Prolog (con el editor SWI – Prolog – Editor 5.6.48) que sirviera de guía para impartir dicha asignatura.

Para llegar a nuestra meta nos enfocamos en diversos puntos, entre los principales fue la elaboración de serie de temas referentes a los conceptos y características más importantes de la Inteligencia Artificial y los Sistemas Expertos.

Establecimos las ventajas que tiene el lenguaje Prolog con respecto al lenguaje Lisp y mencionamos las principales diferencias que tienen entre sí. Es importante mencionar que realizamos un sin número de prácticas que se encuentran relacionadas con cada uno de los temas teóricos, capturando una posible solución a esos problemas propuestos, elaboramos una aplicación donde ponemos en práctica cada uno de los conceptos estudiados y ejecutado en todos los laboratorios.

XI. RECOMENDACIONES

Al concluir nuestro proyecto monográfico recomendamos:

1) Publicar este documento en la página Web del profesor para que los estudiantes y personas interesadas en el tema puedan acceder al mismo desde cualquier lugar.

2) Incentivar a los alumnos a desarrollar nuevos proyectos monográficos en esta asignatura basado en otros lenguajes de programación para una mayor ampliación de la Inteligencia Artificial y Sistemas Expertos.

XII. ANEXOS

ANEXOS

Propuesta de temporización

La asignatura de Inteligencia Artificial y Sistemas Expertos, se imparte en el noveno semestre (primer semestre de quinto año) de Ingeniería en Sistemas de Información.

La asignatura consta de 4 horas semanales (2 teóricas y 2 prácticas) resultando 32 horas teóricas y 32 horas practicas para un total de 64 horas de clases.

Representaremos mediante una tabla dicha temporización para un mejor entendimiento:

Semanas	Clases	Horas	Temas
1	1	2	Presentación de la asignatura
2,3	2,3	4	Inteligencia Artificial y Sistemas Expertos
4,5,6	4,5,6	6	Introducción al lenguaje de programación PROLOG.
7,8,9,10	7,8,9,10	8	Listas en Prolog.
11,12	11,12	4	Estructura de control
13,14	13,14	4	Árboles
15,16	15,16	4	Programación lógica y Base de Dato

Planificación temporal de la parte práctica

Como ya se ha indicado en el apartado anterior, para el desarrollo de las prácticas de laboratorio se cuenta con una sesión de dos horas por semana, para un total de **32 horas** a lo largo de las **16 semanas** del semestre académico**.**

El número de horas asignadas a cada una de las prácticas de laboratorio se establece en base a la complejidad relativa de la misma y al tiempo total disponible, procurando que exista una concordancia entre la teoría y la práctica, de manera que al realizar cada práctica de laboratorio, ya se han impartido los conocimientos teóricos necesarios para poder desarrollarla.

Semanas	Prácticas	Horas	Descripción de la práctica
2	1	2	Descripción e instalación del entorno de desarrollo de SWI- Prolog-Editor
3,4	2	4	Introducción al lenguaje Prolog
5,6	3	4	Entrada y salida
7,8,9,10	4	8	Listas y operaciones en Prolog
11,12	5	4	Estructura de Control
13,14	6	4	Árboles
15,16	7	4	Programación lógica y Bases de dato

Lista de predicados en Prolog

Comandos	Significado
is	Es un operador infijo, que en su parte derecha lleva un término que se interpreta como una expresión aritmética, contrastándose con el término de su izquierda.
real(X)	Se cumple si X representa en ese momento un número real.
Control del programa	
true	El predicado consult esta pensado para leer y compilar un programa Prolog bien para las situaciones en las que se precise añadir las cláusulas existentes en un determinado fichero a las que ya están almacenadas y compiladas en la base de datos. Su sintaxis puede ser una de las siguientes: consult(fichero); consult(fichero.ext); consult('c:\prolog\fichero').
fail	Objetivo que siempre se cumple
consult	Objetivo que siempre fracasa
! Corte	Fuerza al sistema Prolog a mantener ciertas elecciones
repeat	Sirve para generar soluciones múltiples mediante el mecanismo de reevaluación
halt	Salir del sistema Prolog
call(X)	Se cumple si tiene éxito el intento de satisfacer X (instanciada a un término)
recon	El predicado recon es muy parecido a consult, con la salvedad de que las cláusulas existentes en el fichero consultado, reemplazan a las existentes en la base de hechos. Su sintaxis es la misma que la de consult.
Operadores aritméticos y relacionales	
Construcción de objetivos compuestos (conectivas lógicas)	
X=Y	Se cumple si puede hacer X e Y iguales (unificación)
X,Y	Conjunción de objetivos
X;Y	Disyunción de objetivos
X\=Y	Se cumple si X=Y fracasa
not(X)	Se cumple si fracasa el intento de satisfacer X
X==Y	Igualdad más estricta
X\==Y	Se cumple si fracasa ==
Clasificación de términos	
var(X)	Se cumple si X es en ese momento una variable no instanciada
X < Y	Predicado menor que
X > Y	Predicado mayor que
nonvar(X)	Se cumple si X no es una variable sin instanciar en ese momento
X >= Y	Predicado mayor o igual que
atomic(X)	cumple si X representa en ese momento un número o un átomo
=:=	Igualdad aritmética
numeric(X)	Se cumple si X representa en ese momento un número
integer(X)	Se cumple si X representa en ese momento un número entero

Predicados de Entrada y Salida	
forget	Tiene como fin eliminar de la base de datos actual aquellos hechos consultados de un fichero determinado.
get0(X)	Se cumple si X puede hacerse corresponder con el siguiente carácter encontrado en el canal de entrada activo (en código ASCII)
get(X)	Se cumple si X puede hacerse corresponder con el siguiente carácter imprimible encontrado en el canal de entrada activo
exitsys	Este predicado nos devuelve al sistema operativo.
write	El predicado sirve para imprimir en pantalla. Las comillas simples encierran constantes, mientras que todo lo que se encuentra entre comillas dobles es tratado como una lista. También podemos mostrar el valor de una variable, siempre que esté instanciada: write(X).
nl	El predicado nl fuerza un retorno de carro(enter) en la salida.
Funciones aritméticas	
abs(X)	Devuelve el valor absoluto de la expresión X
sign(X)	Devuelve -1 si X < 0, 1 si X > 0 y 0 si X = 0
min(X,Y)	Devuelve el menor de X e Y
max(X,Y)	Devuelve el mayor de X e Y
random(X)	Devuelve un entero aleatorio i (0 =< i < X); la es determinada por el reloj del sistema cuando se arranca SWI-Prolog-Editor
round(X)	Evalúa la expresión X y la redondea al entero más cercano
integer(X)	Evalúa la expresión X y la redondea al entero más cercano
float(X)	Evalúa la expresión X en un número en coma flotante
truncate(X)	Trunca la expresión X en un número entero
floor(X)	Devuelve el mayor número entero menor o igual que el resultado de la expresión X
	Devuelve el menor número entero mayor o igual

ceiling(X)	que el resultado de la expresión X
sqrt(X)	Raíz cuadrada de la expresión X
sin(X)	Seno de la expresión X (ángulo en radianes)
cos(X)	Coseno de la expresión X (ángulo en radianes)
tan(X)	Tangente de la expresión X (ángulo en radianes)
asin(X)	Arcoseno (ángulo en radianes) de la expresión X
acos(X)	Arcocoseno (ángulo en radianes) de la expresión X
atan(X)	Arcotangente (ángulo en radianes) de la expresión X
log(X)	Logaritmo neperiano de la expresión X
log10(X)	Logaritmo en base 10 de la expresión X
exp(X)	e elevado al resultado de la expresión X
pi	Constante matemática pi (3.141593)
Manipulación de la Base de Conocimiento	
listing	Se escriben en el fichero de salida activo todas las cláusulas
listing(X)	Siendo X un átomo, se escriben en el fichero de salida activo todas las cláusulas que tienen como predicado dicho átomo
clause(X,Y)	Se hace coincidir X con la cabeza e Y con el cuerpo de una cláusula existente en la base de conocimiento
assert(X)	Permite añadir la nueva cláusula X a la base de conocimiento
asserta(X)	Permite añadir la nueva cláusula X al principio de la base de Conocimiento
assertz(X)	Permite añadir la nueva cláusula X al final de la base de Conocimiento
retract(X)	Permite eliminar la primera cláusula de la base de conocimiento que empareje con X
retractall(X)	Permite eliminar todas las cláusulas de la base de conocimiento que emparejen con X
abolish(X)	Retira de la Base de Conocimiento todas las cláusulas del predicado X
abolish(X,Y)	Retira de la Base de Conocimiento todas las

	cláusulas del predicado de nombre X y número de argumentos Y
Construcción y acceso a componentes de estructuras	
functor(E,F,N)	E es una estructura con nombre F y número de argumentos N
arg(N,E,A)	El argumento número N de la estructura E es A
E=..L	Predicado univ. L es la lista cuya cabeza es el átomo de la estructura E y la cola sus argumentos
name(A,L)	Los caracteres del átomo A tienen por ASCII los números de la lista L
Depuración de programas Prolog	
trace	Activación de un seguimiento exhaustivo, generando la traza de la ejecución de las metas siguientes
trace(X)	Activación de un seguimiento del predicado X
notrace	Desactiva el seguimiento exhaustivo
notrace(X)	Desactiva el seguimiento mientras dure la llamada al objetivo X
spy(X)	Fijación de puntos espía en el predicado X
nospy(X)	Elimina los puntos espía especificados
nospyall	Elimina todos los puntos espía
debugging	Ver el estado del depurador y los puntos espía se han establecido hasta el momento
debug	Inicia el depurador
nodebug	Desactiva el depurador

Manejo de listas	
is_list(X)	Se cumple si X es una lista o la lista vacía [ ]
append(X,Y,Z)	Z es la concatenación de las listas X e Y
member(X,Y)	X es uno de los elementos de la lista Y
delete(X,Y,Z)	Borra el elemento Y de la lista X y da como resultado la lista Z
select(X,Y,Z)	Selecciona el elemento X de la lista Y y da como resultado la lista Z
nth0(X,Y,Z)	El elemento X-ésimo de la lista Y es Z (empezando en 0)
nth1(X,Y,Z)	El elemento X-ésimo de la lista Y es Z (empezando en 1)
last(X,Y)	X es el último elemento de la lista Y

XIII. REFERENCIAS BIBLIOGRAFÍCAS

1) Aaby, Anthony. Prolog Tutorial http: //cs.wwc.edu/KU/PR/Prolog.html, 5 de Febrero del 1997.

2) Roth, Al. The Practical application of Prolog, Dr. Dobbs Portal http://www.ddj.com/hpc-high-performancecomputing/184405220, 10 de Diciembre 2002.

3) Wikipedia. http: //es.wikipedia.org/wiki/Prolog, 2 de Noviembre 2007.

4) Alonso Jiménez, José. Introducción al programación lógica de Prolog http://www.cs.us.es/~jalonso/publicaciones/2006-int_Prolog.pdf, 17 de Junio 2006.

5) Correas Fernandez, Jesús. Programación lógica Prolog. http://clip.dia.fi.upm.es/~jcorreas/Prolog/master_Prolog_2.pdf, 17 de Junio 2002

6) Rossel, Gerardo. Programación lógica, 2 ed, febrero 2004.

7) http://www.amzi.com/articles/code07_whitepaper.pdf, 15 de Marzo del 2004.

8) Reyes Alfaro Allan, Rugama Escoto William, Ruiz Vado Axel.Soporte para el Componente Currricular de Inteligencia Artificial y Sistemas Expertos basado en Lisp utilizando el editor Xanalys Lispworks 4.3, León, Nic. UNAN, 2007.