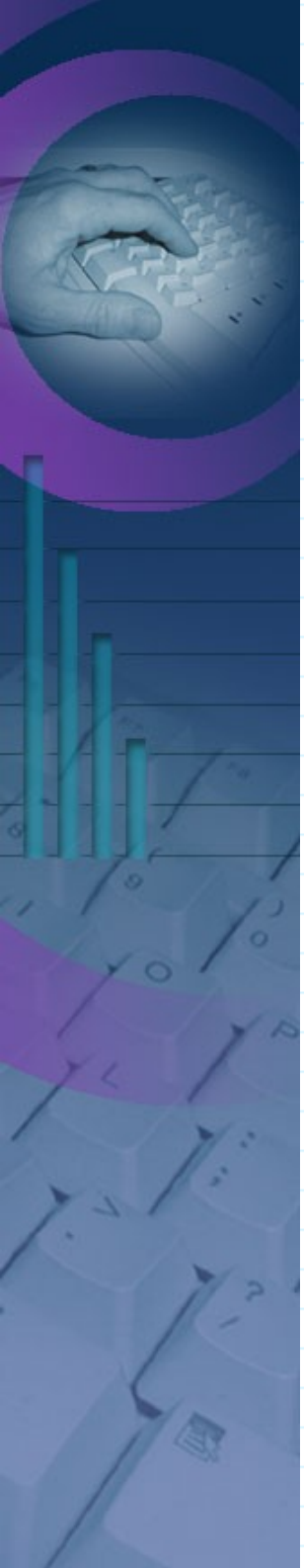

Tecnologías de Programación

Paradigma Lógico – ProLog

Predicados Predefinidos II

Notación Operador

- Prolog utiliza la notación prefija para todos sus predicados.
  - X is +(1, 2). --> X = 3;
- A veces resulta práctico escribir predicados como operadores.
  - X is 1 + 2. --> X = 3;

Notación Operador

- Resulta aún más interesante cuando podemos obtener expresiones como:
  - termo contiene agua.<---> contiene(termo, agua).
  - juan es joven <---> es(juan, joven).
  - maria madre_de juan<---> madre_de(maria, juan).
- En Prolog, el predicado predefinido “:-op/3” nos permite definir predicados como operadores.

Notación Operador

- ":-op/3" permite definir el operador y su forma de uso, entre la información suministrada se encontrará:
  - **Precedencia**: va de 1 a 1200. La máxima corresponde a los últimos operadores que se evaluarán.
  - **Posición**: puede ser infijo (xfx), posfijo (xf) y prefijo(fx).
  - **Asociatividad**: viene dada por la precedencia de los operandos respecto del operador, pueden ser menor (x), o menor o igual (y).

Notación Operador

- Ejemplos de definición de operadores aritméticos:
  - :-op(500, yfx, [+, -]).
  - :-op(500, fx, [+, -]).
  - :-op(400, yfx, [*, /, div]).
- Ejemplo de definición de un operador nuevo con notación posfija:
  - nuevo(auto).
  - :-op(1000, xf, [nuevo]).
    - nuevo(auto). → true
    - auto nuevo. → true

Notación Operador

- El primer argumento define la precedencia, que puede ir de 1 a 1200.
- La precedencia 0 remueve la declaración.
- El segundo argumento define la posición y la asociatividad utilizando las letras f (define la posición del operador), x e y (definen los operandos y la precedencia de los mismos respecto del operador).

Notación Operador

- El tercer argumento define el nombre o lista con los nombres de los predicados definidos como operadores:
  - igual(A, A).
  - distinto(A, B) :- A \= B.
  - :-op(500, xfx, [igual, distinto]).
    - 1 igual 1. -> true.
    - 1 distinto 2. -> true.

Lectura/Escritura y Manejo de Ficheros

- Permiten ingreso y escritura de datos a y desde distintos orígenes.
  - **write(+Term)**: siempre es evalúa como verdadero, escribe el término recibido como argumento en el canal de salida activo.
    - write([1, 2, 3]). → [1, 2, 3]
    - write(1 + 2). → 1 + 2
  - **display(+Term)**: funciona igual que write/1, pero no tiene en cuenta las declaraciones de operadores.
    - display([1, 2, 3]). → .(1, .(2, .(3, [])))
    - display(1 + 2). → +(1, 2)
  - **nl**: escribe un salto de línea.

Lectura/Escritura y Manejo de Ficheros

- **tell(+SrcDest)**: Abre el fichero indicado en su argumento y lo define como canal de salida activo. Si el fichero no existe es creado. Si se utiliza tell/1 con un fichero ya existente, el contenido de dicho fichero se destruye.
  - tell('/tmp/prueba.txt').
- **telling(+SrcDest)**: su argumento unifica con el identificador del stream de salida activo.
  - telling(X). → X = '$stream'(1233816)
- **told**: Cierra el canal de salida activo
- **append(+File)**: Abre un fichero para agregar datos, no destruye su contenido, solo agrega al final.
  - append('/tmp/prueba.txt').

Lectura/Escritura y Manejo de Ficheros

- **see(+SrcDest)**: Abre el fichero indicado en su argumento y lo define como canal de entrada activo. Si el fichero no existe se reporta un error.
  - see('/tmp/prueba.txt').
- **seeing(+SrcDest)**: Se cumple si su argumento coincide con el identificador correspondiente al stream de entrada activo.
  - telling(X). → X = '$stream'(1233816)
- **seen**: Cierra el canal de entrada activo.

Lectura/Escritura y Manejo de Ficheros

- **read(-Term)**: lee un string del canal de entrada activo y lo unifica con la variable indicada en su argumento. Para completar la entrada, se debe ingresar un punto “.” final.

- **put(+Char)**: Escribe en el canal de salida activo el carácter recibido (o cuyo código ASCII se recibe en el argumento).

- **get(-Char)**: Se cumple si su argumento corresponde al siguiente carácter en el canal de entrada activo.

Aritmética

- Algunos predicados para operaciones y comparaciones aritméticas:
  - **between(+Low, +High, ?Value)**: Valida que Value se encuentre entre Low y High. Sus tres argumentos deben ser enteros.
    - Between(1, 3. X).

      → 1;

      → 2;

      → 3.
  - **succ(?Int1, ?Int2)**: valida que Int2 = Int1 + 1.
    - succ(1, 2). → true
    - succ(5, X). → X = 6

Aritmética

- **pluss(?Int1, ?Int2, ?Int3)**: Verdadero si Int3 = Int1 + Int2.
  - pluss(2, 3, 5). → true
  - pluss(2, 3, X). → X = 5.
- **abs(+Exp)**: Evalúa Exp y retorna su valor absoluto.
  - X is abs(-5). → X = 5.
- **sign(+Exp)**: Retorna -1 si Exp < 0, 1 si Exp > 0 y 0 si Exp = 0.
  - X is sign(-2). → X = -1.

Aritmética

- **max(+Exp1, +Exp2)**: retorna el mayor.
  - X is max(5, 3). → X = 5.
- **min(+Exp1, +Exp2)**: retorna el menor.
  - X is min(5, 3). → X = 3.
- **round(+Exp)**: Evalúa Exp y redondea el resultado al entero más próximo.
  - X is round(3.23). → X = 3.
  - X is round(3.73). → X = 4.

Manipulación de Listas

- Algunos predicados para operaciones sobre listas
  - **length(?List, ?Int)**: Verdadero si Int es la cantidad de elementos de la lista List.
    - length([1, 2, 3], X). → X = 3.
  - **sort(+List, -Sorted)**: Verdadero si Sorted es List con los elementos ordenados y sin repeticiones.
    - sort([3, 1, 2, 2, 3], X). → X = [1, 2, 3].
  - **msort(+List, -Sorted)**: igual que sort/2 pero sin eliminar los duplicados.
    - msort([3, 1, 2, 2, 3], X). → X = [1, 2, 2, 3, 3].

Manipulación de Listas

- **append(?List1, ?List2, ?List3)**: Exitoso si List3 unifica con la concatenación de List1 yList2.
  - Append([1, 2], [a, b], X). → X = [1, 2, a, b].

- **member(?Elem, ?List)**: Exitoso cuando Elem puede ser unificado con alguno de los elementos de la lista List.
  - member(2, [1, 2, 3]). → true.
  - member(X, [1, 2, 3]).

    → X = 1;

    → X = 2;

    → X = 3.

Manipulación de Listas

- **delete(+List1, ?Elem, ?List2)**: elimina todos los miembros de List1 que unifiquen con Elem y unifica el resultado con List2.
  - Delete([1, 2, 3], 2, X). → X = [1, 3].
- **last(?List, ?Elem)**: Tiene éxito si Elem unifica con el último elemento de List.
  - Last([1, 2, 3], X). → X = 3.
- **reverse(+List1, -List2)**: Revierte el orden de la lista List1 y unifica el resultado con los elementos de List2.
  - Revert([1, 2, 3], X). → X = [3, 2, 1].

Manipulación de Listas

- **flatten(+List1, -List2)**: Transforma List1 en una lista plana
  - Flatten([1, 2, [3, [4]]], X). → X = [1, 2, 3, 4]
- **max_list(+List, -Max)**: Verdadero si Max es el mayor número resultado de evaluar aritméticamente los elementos de la lista
  - max_list([1, 2, 3, abs(-4), 3, 2], X). → X = 4
- **min_list(+List, -Min)**: Verdadero si Min es el menor número resultado de evaluar aritméticamente los elementos de la lista
  - min_list([1, 2, 3, abs(-4), 3, 2], X). → X = 1