Tecnologías de Programación

Paradigma Lógico – ProLog

Predicados Predefinidos

Predicados Predefinidos

- Los predicados predefinidos son aquellos que ya vienen definidos en Prolog, por lo que no necesitamos especificarlos

- Dos grandes grupos
  - Predicados “comunes” predefinidos en Prolog pero que podríamos definir nosotros tranquilamente
  - Predicados con un efecto colateral distinto a la ligadura de variables a valores

El Esquema Condicional

- En Prolog la conjunción entre dos o más clausulas se expresa separando las mismas con coma (“,”).
  - clausula1, clausula2, ..., clausulaN.
- La disyunción es expresada declarando más de una cláusula para el mismo predicado
  - pertenece(X, [X | _]) :- !.
  - pertenece(X, [_ | L]) :- pertenece(X, L).

El Esquema Condicional

- Otra forma de expresar la disyunción es a través del predicado predefinido punto y coma (“;”)
  - pertenece(X, [Y | L]) :- X = Y, ! ; pertenece(X, L).

El uso del punto y coma (“;”) es equivalente a la declaración de varias cláusulas para el mismo predicado, sin embargo se recomienda acotar su uso por cuestiones de legibilidad

El Esquema Condicional

- La cláusula condicional if-then-else se representa en prolog como “if -> then; else”
  - integer(1) -> write('entero'); write('no entero').
    - entero
  - integer(1.1) -> write('entero'); write('no entero').
    - no entero

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 <---> joven(juan).
  - 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
- 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.

Clasificación de Términos

- Permiten determinar el tipo de término al que nos referimos
  - **var(+Term), novar(+Term)**: se cumplen si Term es una variable no instanciada, y si no lo es, respectivamente
    - var(X). → true
    - var(1). → false
  - **integer(+Term), float(+Term), number(+Term)**: se cumple si X representa un entero, un punto flotante, o un número en general respectivamente
    - integer(1). → true
    - number(1). → true

Clasificación de Términos

- **atom(+Term)**: se cumple si X representa un átomo en Prolog
  - atom(1). → false
  - atom(juan) → true
  - atom(1+1) → false

- **atomic(+Term)**: se cumple si X representa un átomo o un número
  - atomic(1). → true
  - atomic(juan) → true
  - atomic(1+1) → false

Clasificación de Términos

- **ground(+Term)**: se cumple si X no tiene variables libres
  - ground(1 + 2 + 3). → true
  - ground(1 + X + 3). → false
- **is_list(+Term)**: se cumple si Term es una lista
  - is_list([]). → true
  - is_list([1, 2, 3, [a, b]]). → true
  - is_list(1). → false
  - is_list(X). → false

Predicados de Control

- Permiten controlar la evaluación de otros predicados
  - **“!” (corte)**: operador de corte
  - **true, fail**: objetivo que se cumple siempre y fracasa siempre respectivamente
  - **not(+Goal)**: siendo Goal un término que puede interpretarse como objetivo, not(+Goal) se cumple si el intento por satisfacer Goal fracasa
  - **repeat**: siempre es exitoso, provee una forma de insertar infinitos puntos de elección al momento de la evaluación del mismo

Predicados de Control

- **call(+Goal)**: siendo Goal un término que puede interpretarse como objetivo, call(Goal) se cumple si se cumple el intento por satisfacer Goal
  - X = integer(1), call(X). → true
  - X = integer(a), call(X). → false

- **call(+Goal, +ExtraArg1, ...)**: siendo Goal un termino que se pueda invocar y los siguientes argumentos los argumentos de dicho término invocable
  - call(append, [1, 2, 3], [a], X).

    → X = [1, 2, 3, a]

Predicados de Control

- **ignore(X)**: invoca el término en X, y se evalúa verdadero
  - ignore(append([1], [2], X)).
    → X = [1, 2]
  - ignore(append([1], [2], [1])).
    → true
- **“,” (coma)**: especifica una conjunción de objetivos
  - a, b. → a ∧ b
- **“;” (punto y coma)**: especifica una disyunción de objetivos
  - a, b. → a ∨ b

Predicados de Control

- **findall(+Template, :Goal, -Bag)**: Crea una lista de instanciaciones, Template realiza sucesivamente backtracking sobre Goal y unifica el resultado con Bag. Tiene éxito con una lista vacía si Goal no tiene solución
  - f(1).
  - f(2).
  - f(3).
  - findall(g(X), f(X), Y). → Y = [g(1), g(2), g(3)]

- **forall(:Cond, :Action)**: para todas las alternativas posibles al unificar Cond, se evaluará Action
  - forall(f(X), write(X)). → 123

Predicados de Control

- **apply(+Term, +List)**: Agrega los términos en la lista a los argumentos de Term

  - plus(1, 2, X). → X = 3
  - apply(plus, [1, 2, X]). → X = 3

Construcción y Acceso a Componentes de Estructuras

- Permiten construir y manipular componentes compuestos
  - **functor(?T, ?F, ?A)**: Se satisface si T es un término con functor F y aridad A
    - X = punto(1, 2), functor(X, punto, 2). → true
    - X = punto(1, 2), functor(X, punto, 3). → false
  - **arg(?A, +T, ?V)**: T debe estar instanciado como un término y A a un entero entre 1 y la aridad de T. V se unifica con el valor del argumento indicado por A.
    - arg(2, punto(5, 3), X). → X = 3.

Construcción y Acceso a Componentes de Estructuras

- **setarg(+A, +T, +V)**: los argumentos representan lo mismo que en arg/3, pero ahora se asigna el valor V al argumento A
  - X = punto(5, 3), setarg(2, X, 5).

    → X = punto(5, 5).

- **=../2**: se utiliza para construir una estructura dada una lista de argumentos, el primer argumento representa el functor y el resto los argumentos del mismo
  - X =..[punto, 5, 3].

    → punto(5, 3)

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

- **read(-Term)**: lee un término 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 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

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 sino que 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

Manipulación de Base de Datos

- Prolog ofrece mecanismos para manipular la base de conocimiento en forma dinámica
  - **:-dynamic(X)**: se utiliza para especificar que un predicado en particular se manipulara dinámicamente. X se expresa en la forma "predicado" o "predicado/aridad"
    - :-dynamic(f/1).
  - **asserta(+Term) / assertz(+Term)**: Añade cláusulas al inicio/final del conjunto de cláusulas en la BD que tienen el mismo nombre de predicado.
    - asserta(f(1)).
    - assertz(f(1)).

Manipulación de Base de Datos

- **retract(+Term)**: Elimina de la base de conocimientos las cláusulas que unifican con el átomo ingresado como argumento
  - retract(f(2)).
- **retractall(+Head)**: Elimina de la base de conocimientos las cláusulas cuya cabecera unifican con el átomo ingresado como argumento
  - retractall(f(X)).

Manipulación de Base de Datos

- **listing**: lista todas las cláusulas definidas en la base de conocimiento

- **listing(+Pred)**: lista todas las cláusulas definidas en la base de conocimiento que unifican con el predicado que se pasa como parámetro. El parámetro pasado puede ser de la forma "predicado" o "predicado/aridad"
  - listing(f(X)).
  - listing(f/1).

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 en List
  - max_list([1, 2, 3, 4, 3, 2, 1], X). → X = 4
- **min list(+List, -Min)**: Verdadero si Min es el menor número en List
  - min_list([1, 2, 3, 4, 3, 2, 1], X). → X = 1