Paradigmas de la programación

Diego D. Quiros Vicencio

Manual de reportes.

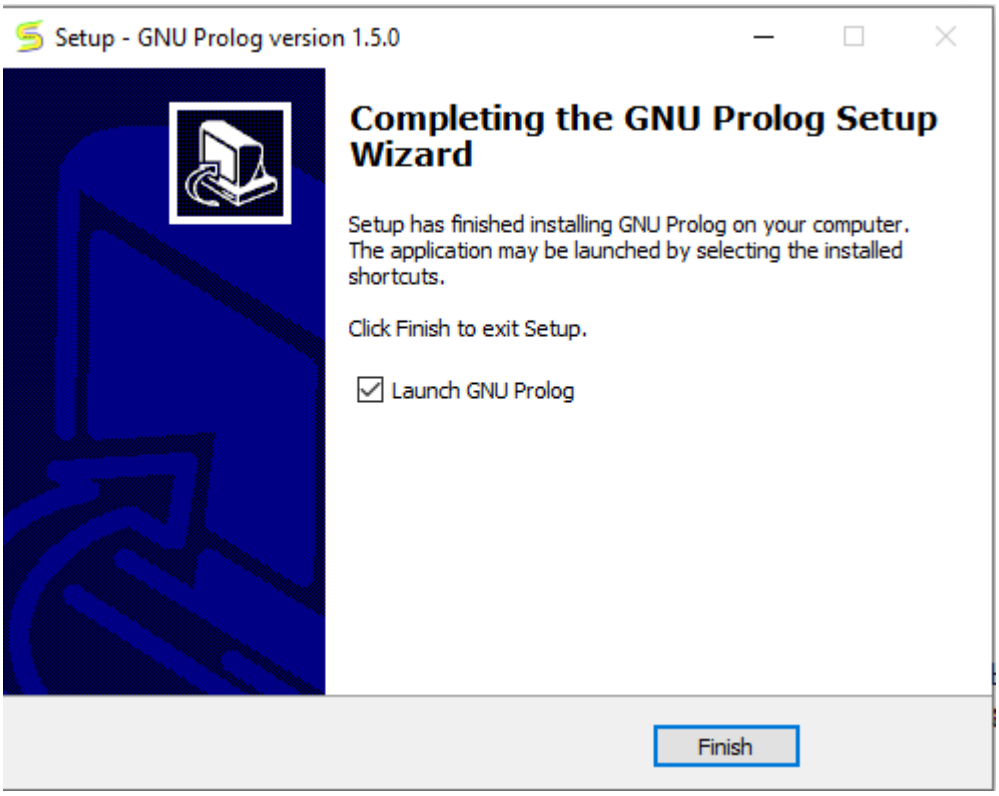
Fecha de Entrega: 30 de mayo de 2024.

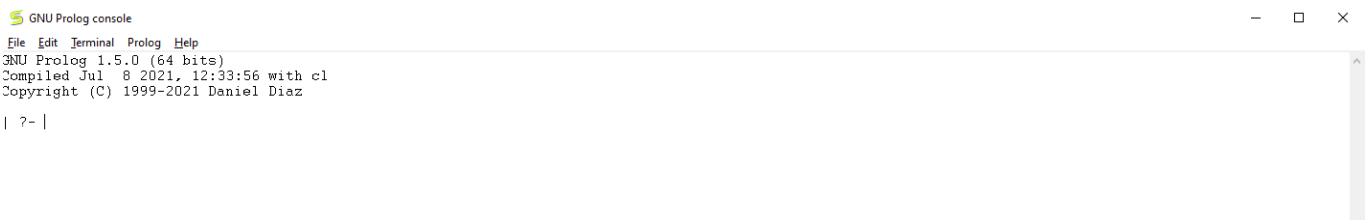
Reporte Practica 4 de la Clase Paradigmas de Programacion

Alumno Diego Demian Quiros Vicencio - 372688

Introduccion a Prolog

Primer Paso instalacion de Prolog

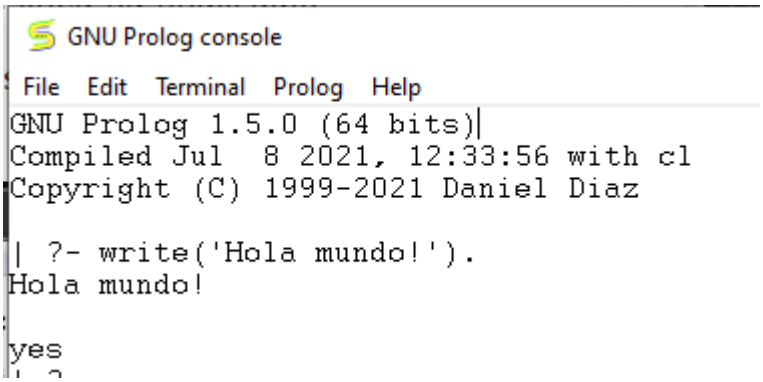




Hola Mundo! en prolog

Podemos escribir "Hola mundo!" desde la consola escribiendo el siguiente codigo:

```
write('Hello World').
```



Base de conocimientos

Esta es una de las partes fundamentales de la programación lógica. Veremos en detalle sobre la Base de Conocimiento y cómo ayuda en la programación lógica.

Hechos, reglas y consultas

Estos son los componentes básicos de la programación lógica. Obtendremos un conocimiento detallado sobre hechos y reglas, y también veremos algunos tipos de consultas que se utilizarán en la programación lógica.

Hechos

Podemos definir el hecho como una relación explícita entre objetos y las propiedades que estos objetos podrían tener. De modo que los hechos son incondicionalmente verdaderos por naturaleza. Supongamos que tenemos algunos hechos como se detallan a continuación:

- tom es un gato
- A Kunal le encanta comer pasta.
- el pelo es negro
- A Nawaz le encanta jugar.
- Pratyusha es vaga.

La sintaxis de los hechos es la siguiente:

```
relation(object1,object2...).
```

El siguiente es un ejemplo del concepto anterior:

```
cat(tom).
loves_to_eat(kunal,pasta).
of_color(hair,black).
loves_to_play_games(nawaz).
lazy(pratyusha).
```

Reglas

Podemos definir la regla como una relación implícita entre objetos.

Consultas

Son algunas preguntas sobre las relaciones entre objetos y propiedades de los objetos. Entonces, la pregunta puede ser cualquier cosa, como se indica a continuación:

- ¿Tom es un gato?
- ¿A Kunal le encanta comer pasta?
- ¿Lili está feliz?
- ¿Ryan irá a jugar?

Entonces, de acuerdo con estas consultas, el lenguaje de programación Logic puede encontrar la respuesta y devolverlas.

Relaciones

Hay varios tipos de relaciones, algunas de las cuales también pueden ser reglas. Una regla puede descubrir una relación incluso si la relación no está definida explícitamente como un hecho.

Objetos de datos

Tipos de Objetos de Datos

1. Atomos: Nombres constantes como tom, pat.
2. Números: Enteros y reales, por ejemplo, 100, 3.14159
3. Variables: Cadenas que comienzan con una letra mayúscula o _, como X, _var.

Atomos

- Reglas:
    - Letras minúsculas, dígitos, y _.
    - Ejemplos: mediodía, b_59.
    - Cadenas entre comillas simples: 'Ropa'.

Numeros

- Enteros: 100, -81.
- Reales: 3.14159, -0.00062.

Variables

- Definidas con letras mayúsculas o _.
- Ejemplos: X, _a50.

Variables Anonimas

- Representadas por _.
- Cada variable anónima es única.

Operadores de Comparación

Los operadores de comparación se utilizan para comparar dos expresiones o estados.

```
Operador    Significado
X > Y       X es mayor que Y
X < Y       X es menor que Y
X >= Y      X es mayor o igual que Y
X =< Y      X es menor o igual que Y
X =:= Y     X    y Y tienen el mismo valor
X =\= Y     X y Y no tienen el mismo valor
```

Operadores Aritméticos

Los operadores aritméticos realizan operaciones matemáticas.

```
Operador    Significado
+           Suma
-           Resta
*           Multiplicación
/           División
**          Potencia
//          División entera
mod         Módulo
```

Bucles en Prolog

En Prolog, los bucles se implementan utilizando recursión, no hay bucles directos como en otros lenguajes. Aquí hay un ejemplo de un bucle recursivo:

```
count_to_10(10) :- write(10), nl.
count_to_10(X) :-
   write(X), nl,
   Y is X + 1,
   count_to_10(Y).
```

Toma de Decisiones

Prolog usa declaraciones If-Then-Else. Ejemplo:

```
gt(X, Y) :- X >= Y, write('X is greater or equal').
gt(X, Y) :- X < Y, write('X is smaller').
```

```
gte(X, Y) :- X > Y, write('X is greater').
gte(X, Y) :- X =:= Y, write('X and Y are same').
gte(X, Y) :- X < Y, write('X is smaller').
```

Conjunción

La conjunción (lógica AND) se implementa con el operador coma ,.

Disyunción

La disyunción (lógica OR) se implementa con el operador punto y coma ;

Listas

La lista es una estructura de datos simple que se usa ampliamente en programación no numérica. La lista consta de cualquier número de elementos, por ejemplo, rojo, verde, azul, blanco y oscuro. Se representará como [rojo, verde, azul, blanco, oscuro]. La lista de elementos irá entre corchetes.

Una lista puede estar vacía o no vacía. En el primer caso, la lista se escribe simplemente como un átomo de Prolog: []. En el segundo caso, la lista consta de dos componentes:

- El primer elemento, denominado cabeza de lista.
- La parte restante de la lista, llamada cola.

Supongamos que tenemos una lista como [rojo, verde, azul, blanco, oscuro]. Aquí, la cabeza es rojo y la cola es [verde, azul, blanco, oscuro]. Entonces, la cola es otra lista.

Operaciones básicas en listas

```
Operaciones                 Definición
Comprobación de membresía   Verificar si un elemento determinado es miembro de una
lista especificada.
Cálculo de longitud         Encontrar la longitud de una lista.
Concatenación               Unir/agregar dos listas.
Eliminar objetos            Eliminar el elemento especificado de una lista.
Agregar elementos           Agregar una lista a otra (como un elemento).
Insertar elementos          Insertar un elemento determinado en una lista.
```

Recursividad

La recursividad en Prolog permite que un predicado se llame a sí mismo, posiblemente con modificaciones en sus argumentos, para resolver problemas de manera iterativa.

Estructuras

Las estructuras en Prolog son objetos de datos que contienen múltiples componentes.

Coincidencia

La coincidencia de patrones se usa para verificar si dos términos son idénticos o si las variables en ambos términos pueden referirse a los mismos objetos después de instanciarlas.

Árboles Binario

Se presenta un ejemplo de árbol binario utilizando estructuras recursivas en Prolog.

```
node(2, node(1,nil,nil), node(6, node(4,node(3,nil,nil), node(5,nil,nil)),
node(7,nil,nil))).
```

Backtracking

El backtracking es un procedimiento en el que Prolog busca el valor de verdad de diferentes predicados verificando si son correctos o no. Proceso: Comienza desde un punto inicial y explora diferentes caminos hasta que encuentra la solución o destino adecuado. Si se encuentra un callejón sin salida (una solución que no cumple con las condiciones), Prolog vuelve atrás para explorar otras posibilidades.

Precauciones con el backtracking

- Ineficiencia potencial: A veces, el backtracking puede ser ineficiente si se ejecuta sin control, especialmente en programas que tienen reglas recursivas o sistemas de toma de decisiones.
- Solución: Se introdujo el operador de corte (!) en Prolog para controlar el backtracking. El corte se utiliza para eliminar alternativas que sabemos que no conducirán a una solución válida, mejorando así la eficiencia del programa.

Predicado different (diferente)

El predicado different(X, Y) se utiliza para determinar si dos argumentos dados son diferentes entre sí. Si son iguales, el predicado falla (devuelve false), y si son diferentes, tiene éxito (devuelve true). Hay dos formas de implementar este predicado en Prolog: usando clausulas disyuntivas y separadas

Predicado not (no)

El predicado not(P) se utiliza para negar un enunciado en Prolog. Si el enunciado P es verdadero, not(P) falla (devuelve false), y si P es falso, not(P) tiene éxito (devuelve true).

Manejo de Entradas y Salidas

Predicado write()

El predicado write() se utiliza para escribir en la salida estándar o en un archivo.

Predicado read()

El predicado read() se utiliza para leer desde la entrada estándar o desde un archivo.

Manipulación de Archivos

Predicados tell y told

- tell('filename'): Abre el archivo filename para escritura.
- told.: Cierra el archivo actual.

Predicados see y seen

see('filename'): Abre el archivo filename para lectura. seen.: Cierra el archivo actual.

Manipulación de Caracteres

Predicados put y put_char

- put(C): Escribe el carácter ASCII C.
- put_char(C): Escribe el carácter C.

Identificación de Términos

```
Predicado   Descripción
var(X)      Verdadero si X es una variable no instanciada.
nonvar(X)   Verdadero si X no es una variable o está instanciada.
atom(X)     Verdadero si X es un átomo.
number(X)   Verdadero si X es un número.
integer(X)  Verdadero si X es un entero.
float(X)    Verdadero si X es un número real.
atomic(X)   Verdadero si X es un átomo o un número.
compound(X) Verdadero si X es una estructura (término complejo).
ground(X)   Verdadero si X no contiene variables no instanciadas.
```

Descomposición de Estructuras

```
Predicado                    Descripción
functor(T, F, N)           Verdadero si F es el principal functor del término T
con aridad N.
arg(N, Term, A)            Verdadero si A es el N-ésimo argumento del término
compuesto Term.
..(Term, L) Term =.. L.    Verdadero si L es una lista que contiene el functor y
sus argumentos.
```

Predicados Matemáticos

Prolog también proporciona un conjunto de predicados matemáticos:

```
Predicado              Descripción
random(L, H, X)        Obtiene un valor aleatorio entre L y H.
```

```
between(L, H, X)        Obtiene todos los valores entre L y H.
succ(X, Y)              Añade 1 a X y unifica con Y.
abs(X)                  Obtiene el valor absoluto de X.
max(X, Y)               Obtiene el valor más grande entre X e Y.
min(X, Y)               Obtiene el valor más pequeño entre X e Y.
round(X)                Redondea X al entero más cercano.
truncate(X)             Trunca X (convierte float a entero).
floor(X)                Redondea hacia abajo X.
ceiling(X)              Redondea hacia arriba X.
sqrt(X)                 Obtiene la raíz cuadrada de X.
```

Implementación de la Estructura de Árbol en Prolog

Definición de la base de datos del árbol

```
:- op(500, xfx, is_parent).

/* Definición de las relaciones padre-hijo */
a is_parent b.
c is_parent g.
f is_parent l.
j is_parent q.
a is_parent c.
c is_parent h.
f is_parent m.
j is_parent r.
a is_parent d.
c is_parent i.
h is_parent n.
j is_parent s.
b is_parent e.
d is_parent j.
i is_parent o.
m is_parent t.
b is_parent f.
e is_parent k.
i is_parent p.
n is_parent u.
n is_parent v.
```

Definición de operadores

```
:- op(500, xfx, is_sibling_of).

/* X e Y son hermanos si comparten el mismo padre y son diferentes */
X is_sibling_of Y :- Z is_parent X,
                     Z is_parent Y,
                     X \== Y.
```

```
/* Un nodo es una hoja si no tiene hijos */
leaf_node(Node) :- \+ is_parent(Node, _).

:- op(500, xfx, is_at_same_level).

/* X e Y están en el mismo nivel si tienen el mismo padre */
X is_at_same_level X.
X is_at_same_level Y :- W is_parent X,
                        Z is_parent Y,
                        W is_at_same_level Z.
```

Consultas Ejemplo

```
| ?- consult('tree_data.pl').
% tree_data.pl compiled 0.00 sec, 0 clauses
true.

| ?- i is_parent p.
yes.

| ?- i is_parent s.
no.

| ?- is_parent(i, p).
yes.

| ?- e is_sibling_of f.
yes.

| ?- is_sibling_of(e, g).
no.

| ?- leaf_node(v).
yes.

| ?- leaf_node(a).
no.

| ?- is_at_same_level(l, s).
yes.

| ?- l is_at_same_level v.
no.
```

Operaciones Avanzadas sobre la Estructura de Árbol Encontrar la Altura de un Nodo

```
/* Definir la altura de un nodo */
ht(Node, H) :- leaf_node(Node), H is 0.
ht(Node, H) :- is_parent(Node, Child),
```

```
                ht(Child, ChildHeight),
                H is ChildHeight + 1.

/* Encontrar la altura máxima de un árbol */
max([X|R], M) :- max(R, X, M).
max([], M, M).
max([X|R], M0, M) :- X > M0, max(R, X, M).
max([X|R], M0, M) :- X =< M0, max(R, M0, M).

/* Calcular la altura de un árbol */
height(N, H) :- setof(Z, ht(N, Z), Set),
                max(Set, 0, H).
```

Consultas de Altura

```
| ?- height(a, H).
H = 3.

| ?- height(b, H).
H = 2.

| ?- height(n, H).
H = 1.
```