Intelligente Systemen Programmeeropdracht 2: Predicaten definiëren in Prolog

Deadline: zaterdag 13 maart, 23:59 uur

(Deze versie is van 5 maart 2021, 03:23)

Je moet deze programmeeropdracht **zelf** en **alleen** maken. Je mag met anderen overleggen over de voors en tegens van verschillende oplossingsstrategieën en datastructuren, want daar leer je allebei van! Maar je mag niet je eigen code laten zien of met anderen delen, en ook geen code van anderen overnemen. Je mag ook geen code van internet overnemen. De uiterst simpele algemene regel is:

> *Elke letter code die je submit moet door jou zelf bedacht en geschreven zijn.*

Het is (verstandiger maar vooral) leuker `<preek>`en leerzamer`</preek>` om je eigen code te maken en je eigen "CORRECT" te verdienen op DOMjudge. Je moet je programma schrijven in Prolog, en inleveren via DOMjudge.

Inhoudsopgave

1 Opdrachtbeschrijving

Lees allereerst de handleiding voor het submitten van Prolog code op DOMjudge, die je kunt vinden op Blackboard. Er staat dingen in die je moet doen en dingen die je moet laten en als je je daaraan houdt, bespaar je ons allemaal tijd en jezelf stress.

Voor deze opdracht moet je een aantal predicaten definiëren door Prolog facts en rules te schrijven. Omdat de *practical sessions* volgens de auteurs van LPN (en mij) de belangrijkste onderdelen van het hele boek zijn, wil ik je aansporen hiermee te oefenen door je voor deze practicumopdracht een aantal van deze opdrachten en LPN-exercises te laten doen, naast een paar andere predicaten. Je moet alle **predicaten precies zo noemen als in deze opdracht**, want anders kan ik ze in mijn testcases niet aanroepen.

Ik heb deze tweede opdracht verdeeld in een aantal deelopdrachten. Voor elke deelopdracht schrijf je de definitie van een of meerdere predicaten door facts en rules te specificeren.

Het bestand waarin je dat doet submit je bij het juiste 'problem' van DOMjudge, zodat dat predicaat los van andere predicaten wordt getest op een aantal testcases. Hierdoor krijg je gerichtere feedback van DOMjudge, dan wanneer je alle predicaten in één bestand samenvoegt en die in DOMjudge allemaal tegelijk worden getest.

Instructies (beperkingen) Hoewel je je eigen code moet bedenken en programmeren, kun je niet om bepaalde ingebouwde predicaten van Prolog heen. Maar je mag niet onbeperkt gebruik maken van hetgeen Prolog biedt. Wat níet mag, is "`;`" als "of" gebruiken, want daar wordt code minder goed leesbaar door. Je mag ook geen cut "`!`" gebruiken. Je mag natuurlijk wél `=/2` gebruiken, en ook `\=/2`, en in het algemeen mag je de negatie `\+/1` gebruiken. Je mag Prolog laten rekenen, en dus gebruik maken van het predicaat `is/2`, en van rekenkundige functies zoals `+/2`, `-/2`, etc., en operatoren zoals `</2`, `=</2`, `>/2`, en `>=/2`. Soms is het handig om te weten of een variabele aan een getal gebonden is, daar mag je een ingebouwd predicaat voor gebruiken. Uiteraard mag je ook lijsten `[H|T]` gebruiken. Om PR2 tot een goed einde te brengen, is dit afdoende. Als je andere relaties nodig hebt (dan hieronder genoemd staan), dan moet je die zelf definiëren. Je mag dan ook predicaten gebruiken die je (zelf!) bij een andere deelopdracht hebt geschreven.

2 Deelopdrachten

Ik weet dat je met weinig moeite online een aantal van de oplossingen kunt vinden (de crossword van sectie 2.1 bijvoorbeeld hier en hier), maar probeer het alsjeblieft zelf. Het is echt veel leuker om dit uit te vogelen dan over te tikken van een website. Het DOMjudge probleem waar je je code moet submitten staat in de sectietitel tussen haakjes.

2.1 Crossword ('2.1 cross')

Schrijf een definitie van het predicaat `crossword/6` dat in de exercise 2.4 van LPN wordt beschreven. Er mogen verschillende oplossingen zijn, want DOMjudge kent alle mogelijke oplossingen voor een gegeven set woorden.

2.2 Travel ('2.2 travel')

Hier moet je een definitie geven van het predicaat `travel/3` zoals beschreven in **onderdeel 3** van de practical session van hoofdstuk 3 van LPN. Het gaat dus om een predicaat dat als resultaat (als dat bestaat) een substitutie van de variabele `X` geeft in de gegeven vorm, alleen *zonder line breaks en formattering*—dat is makkelijker. Je hoeft dus niet de reisvorm te bepalen, zoals in onderdeel 4 van deze practical session. Voor de gegeven data in de opdracht, zou Prolog op de query

```
?- travel(metz, losAngeles, Route).
```

als resultaat

```
Route = go(metz, paris, go(paris, losAngeles))
```

moeten vinden. Maar nogmaals: voeg geen facts als `byCar(valmont,metz).` toe aan je sourcecode!

2.3 Binaire bomen spiegelen ('2.3 spieg')

In een *volledige binaire boom* ('*full binary tree*') heeft elke interne knoop precies twee kinderen (zie Cormen, Leiserson, Rivest & Stein, "Introduction to Algorithms", p. 1178). Een 'leaf' knoop is de kleinste binaire boom, met 0 kinderen. We representeren een leaf met de term `leaf(Naam)`, waar `Naam` de naam van de leaf is, en een binaire boom met de term `tree(B1,B2)`, waar `B1` en `B2` binaire bomen zijn (dus recursief ook leaves of binaire bomen). Schrijf een Prolog predicaat `spiegel/2`, zodat `spiegel(B1,B2)` slaagt desda `B2` het spiegelbeeld is van `B1`, zoals in:

```
?- spiegel(tree(leaf(a),tree(tree(leaf(b),leaf(c)),leaf(d))),T).
T = tree(tree(leaf(d), tree(leaf(c), leaf(b))), leaf(a)).
```

Als je deze bomen op papier tekent, begrijp je wat er bedoeld wordt. Dit is exercise 3.5 uit LPN, maar als je die al gemaakt hebt, moet je de naam van het predicaat veranderen voordat je je code submit op DOMjudge. (Het daar gevraagde predicaat heet `swap/2`.)

2.4 Palindroom ('2.4 lepel')

Je gaat een definitie schrijven van een palindroom-predicaat, met een lijst als enige argument. Dit predicaat moet waar zijn desda het argument een palindroom is, dus een lijst met termen die van links naar rechts hetzelfde is als van rechts naar links. De elementen zélf hoeven geen palindroom te zijn, dus `[a,bc,def,bc,a]` is een palindroom. Je moet twee verschillende definities van dit predicaat geven, dus het worden eigenlijk twee predicaten—met verschillende namen, want ze hebben hetzelfde aantal argumenten (zie hieronder).

De eerste manier maakt gebruik van het (door jou te schrijven) predicaat `plak/3` met als argumenten drie lijsten, zodat `plak(L1,L2,L3)` slaagt desda `L3` gelijk is aan de elementen van lijst `L1`, gevolgd door de elementen van lijst `L2`. Schrijf dus eerst dit predicaat `plak/3`. (Het is de 'desda' versie van het ingebouwde predicaat `append/3`.) We willen dus bijvoorbeeld het volgende.

```
?- plak([1,2], [2,3], [1,2,2,3]).
true
?- plak([1,2,3], [4,5], X).
X = [1,2,3,4,5].
?- plak(X,[2,3,4],[1,2,3,4]).
X = [1].
```

De tweede manier maakt gebruik van het predicaat `omgedraaid/2` met als argumenten twee lijsten, zodat `omgedraaid(X,Y)` slaagt desda `Y` een lijst is die gelijk is aan `X`, maar dan van achteren naar voren. De naïeve implementatie van `omgedraaid/2` is left recursive, dus jouw code moet intern een predicaat `omgedraaid/3` gebruiken (dat je dus ook moet schrijven), en dat een accumulator gebruikt. Wat er dus moet gebeuren is o.a. het volgende.

```
?- omgedraaid([1,2,3],[3,2,1]).
true
?- omgedraaid([0,1],X).
X = [1,0].
?- omgedraaid([0,1],[1,1]).
false
```

Ik weet dat de definities van deze predicaten in LPN staan. Ik raad je sterk aan zelf te bedenken hoe dit moet, of in elk geval een definitie (opnieuw) te bedenken zonder in LPN te spieken naar wat daar staat.

Je moet nu een definitie van het predicaat `palindroomPlak/1` schrijven die gebruik maakt van het predicaat `plak/3`. Schrijf daarna een definitie van het predicaat `palindroomDraai/1` die gebruik maakt van het predicaat `omgedraaid/2`. Er moet natuurlijk gelden dat `palindroomPlak(P)` en `palindroomDraai(P)` slagen desda `P` een 'palindroom-lijst' is. Upload al je predicaten in hetzelfde bestand, ze zullen ook afzonderlijk worden getest.

2.5 Fibonacci ('2.5 fib')

Je kunt Prolog laten rekenen, zoals we hier zullen onderzoeken door haar Fibonaccigetallen te laten uitrekenen. Er zijn verschillende manieren waarop je de reeks kunt laten beginnen. We gebruiken hier (en op DOMjudge) $F_1 = F_2 = 1$, en $F_n = F_{n-1} + F_{n-2}$ voor $n \geq 3$. Er is een groot verschil tussen de berekeningstijd van een left-recursieve versus een tail-recursieve definitie van het predicaat `fibonacci/2`. Je moet beide definities geven, dus één zonder en één met accumulator, `fibonacciL/2` en `fibonacciT/2`, respectievelijk. (De `L` en `T` staan voor Left respectievelijk Tail.) De goals `fibonacciL(N,F)` en `fibonacciT(N,F)` moeten slagen desda `F` het `N`-de Fibonacci getal is. De left recursive versie begint (op mijn hagelnieuwe laptop) bij $n = 28$ merkbaar trager te worden, heeft verschillende seconden nodig om de waarde $F_{32} = 2\,178\,309$ te berekenen, en geeft bij $n = 33$ de melding "`Stack limit (1.0Gb) exceeded`". De tail recursive versie daarentegen geeft in een oogwenk de waarde F_{30}, maar in dezelfde oogwenk ook de waarden

$$F_{100} = 354\,224\,848\,179\,261\,915\,075$$

en

$$\begin{aligned} F_{1000} = {} & 43\,466\,557\,686\,937\,456\,435\,688\,527\,675\,040\,625\,802\,564\,660\,517\,371\,780\,402 \\ & 481\,729\,089\,536\,555\,417\,949\,051\,890\,403\,879\,840\,079\,255\,169\,295\,922\,593\,080 \\ & 322\,634\,775\,209\,689\,623\,239\,873\,322\,471\,161\,642\,996\,440\,906\,533\,187\,938\,298 \\ & 969\,649\,928\,516\,003\,704\,476\,137\,795\,166\,849\,228\,875 \end{aligned}$$

Behold the power of recursing tailwise!