Logisch programmeren 2012

Opdrachten Week 2

1

Maak de oefeningen bij hoofdstuk 3 van *LPN*, bij hoofdstuk 5 de oefeningen die niet van lijsten gebruik maken. Check je antwoorden met de Oplossingen achter in het boek. Deze opdracht hoef je niet in te leveren.

2 Successor getallen

In de opdrachten van deze sectie werken we met de successor representatie van natuurlijke getallen. Voor de natuurlijke getallen $\mathbb{N}$ representeren we 0 als `0`, 1 als `s(0)`, 2 als `s(s(0))`, etc. Hieronder de recursieve definitie van natuurlijke getallen:

```
nat(0).
nat(s(X)) :- nat(X).
```

Optellen kan dan met een predicaat `add/3` recursief geformuleerd worden als volgt:

```
add(0,Y,Y) :- nat(Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Weglaten van de conditie `nat(Y)` in de niet-recursieve zin voor `add/3` zou ertoe leiden dat bijvoorbeeld `add(0,mia,mia)` slaagt. In de rest van deze sectie gaan we er van uit dat we bij het aanroepen van onze rekenpredicaten alleen van natuurlijke getallen (in successor notatie) gebruik maken. Gegeven die aanname kunnen we de conditie `nat(Y)` weglaten.

2.1

Definieer de relatie '$\leq$' voor successorgetallen. Gebruik daarvoor een predicaat `leq/2`. `leq(X,Y)` slaagt als `X` en `Y` natuurlijke getallen zijn waarbij `X` kleiner dan of gelijk aan `Y` is. Je definitie is recursief. Stel je dus de volgende vragen: Hoe ziet het grensgeval eruit? Hoe werkt het recursieve geval toe naar het grensgeval?

2.2

Vermenigvuldigen kunnen we zien als herhaald optellen:

$$n \times a = \underbrace{a + \ldots + a}_{n}$$

Hieronder de recursieve definitie van vermenigvuldiging voor natuurlijke getallen in successor notatie die deze gedachte uitwerkt.

$$\begin{aligned} 0 \times a &= 0 \\ s(n) \times a &= (n \times a) + a \end{aligned}$$

We kunnen deze definitie zo omschrijven tot een programma voor `times/3`. Voor natuurlijke getallen `X, Y, Z` slaagt `times(X,Y,Z)` als `Z` het product is van `X` en `Y`.

```
times(0,_,0).
times(s(X),Y,Z) :- times(X,Y,W),add(W,Y,Z).
```

Opdracht Zoals we hierboven vermenigvuldigen zien als herhaald optellen, kunnen we ook machtsverheffing zien als herhaald vermenigvuldigen:

$$a^n = \underbrace{a \times \ldots \times a}_{n}$$

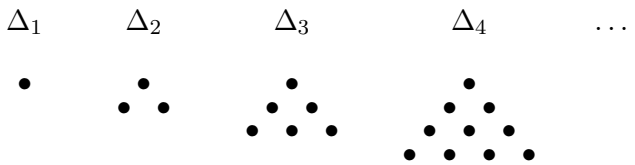
Definieer een predicaat `exp/3` met de volgende interpretatie: voor natuurlijke getallen `X, Y, Z` slaagt `exp(X,Y,Z)` als `Z` het resultaat is van `Y` tot de macht `X`.

Denk goed na over de grensgevallen van je definitie: dat zijn de gevallen waar de exponent 1 of 0 is. Je definitie hoeft niet te voorzien in het geval 0^0 – over de uitkomst daarvan zijn de geleerden het niet eens.

3 Figuurgetallen

3.1

Hieronder het begin van de reeks *driehoeksgetallen*: 1,3,6,10,... Die reeks geeft het aantal biljartballen dat je nodig hebt om een gelijkzijdige driehoek te vullen. Definieer een predicaat `drie/2`. Schrijf je definitie voor getallen in successor notatie: `drie(Zijde,Ballen)` slaagt, als `Ballen` het aantal biljartballen is dat je nodig hebt om een driehoek met zijde `Zijde` te vullen. Bijvoorbeeld: `drie(s(s(0)),s(s(s(0)))` voor Δ_2 hieronder.



Je predicaat `drie/2` is recursief. Denk dus weer goed na: hoe ziet het grensgeval eruit? hoe ziet de recursieve regel eruit? Maak gebruik van `add/3`.

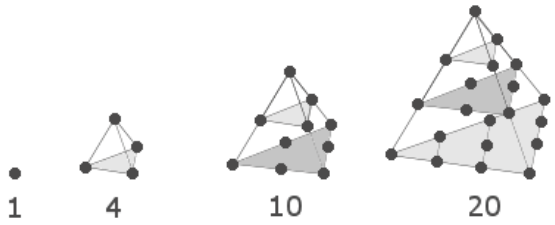
3.2

Geef een programma `drie_dec/2` waarmee je de driehoeksgetallen nu berekent aan de hand van de Prolog rekenpredicaten van Hoofdstuk 5.

Opgelet Bij gebruik van de successor notatie kon je het idee 'natuurlijk getal groter dan nul' uitdrukken via pattern matching: `s(X)`. Bij gebruik van de decimale notatie zal je dat idee in een expliciete conditie `N>0` moeten vertalen.

3.3

De reeks van *piramidegetallen* is de $3D$ variant van de driehoeksgetallen: 1, 4, 10, 20, 35, 56, 84, 120, ... Het n-de piramidegetal geeft aan hoeveel sinaasappels je nodig hebt om een piramide met driehoekig grondvlak te bouwen met een zijde van lengte n.



Definieer een predicaat `tetra/2`: `tetra(N,M)` slaagt als `M` het `N`-de piramidegetal is. Gebruik decimale getallen en de bijhorende Prolog rekenpredicaten.

4 Accumulatoren

Kijken we nog eens naar onze definitie van vermenigvuldigen `times/3` uit §2.2.

```
times(0,_,0).
times(s(X),Y,Z) :- times(X,Y,W),add(W,Y,Z).
```

Het programma is niet erg efficiënt. De recursieve aanroep van `times/3` is de eerste body literal: er wordt niets uitgerekend tot `times/3` bij het grensgeval `times(0,_,0)` is aanbeland. Zet `trace/0` aan, en volg de stappen van de berekening voor een simpel voorbeeld, of teken een bewijsboom.

Voor een iteratieve programmeerstijl kan je in Prolog gebruik maken van een *accumulator* variabele, die de tussentijdse resultaten van de iteratie doorgeeft. Zie het programma voor `times/4` hieronder. `times(X,Y,Acc,Product)` rekent het `Product` uit van `X` en `Y` met behulp van accumulator `Acc`.

```
times(s(X),Y,A,Z) :- add(Y,A,A1),times(X,Y,A1,Z).
times(0,_,A,A).
```

Voor de initiële aanroep van `times/4` stellen we de accumulator gelijk aan `0`. Elke stap van de recursie maakt een tussentijdse som aan en geeft die door via de accumulator variabele. Als het grensgeval wordt bereikt (vermenigvuldigen met zero) is de accumulator gelijk aan de gezochte eindwaarde van het product. De recursieve zin van de definitie heeft nu de aanroep van `times/4` als *laatste* onderdeel: we spreken dan van *staartrecursie*.

Je kan een *wrapper* maken, waarmee je `times/3` definieert in termen van `times/4`. De wrapper definitie vult de goede startwaarde voor de accumulator alvast in.

```
times(X,Y,Product):-times(X,Y,0,Product).
```

Opdracht Herformuleer je eerdere definities van `exp/3` uit §2.2, `drie_dec/2` uit §3.2 en `tetra/2` uit §3.3 tot staartrecursieve definities `exp/4`, `drie/3` en `tetra/3` die gebruik maken van een accumulator. Schrijf de bijhorende wrappers `exp_accu/3`, `drie_accu/2` en `tetra_accu/2` waarbij je de goede startwaarde voor de accumulator meegeeft.

□