Logisch programmeren 2012

Opgaven Week 7

1 Cut, negatie, structuur van termen

1.1

Eén van de sorteeralgoritmen werkt als volgt: zoek in de te sorteren lijst twee opeenvolgende elementen die niet in de goede volgorde staan en wissel die om; herhaal dit proces voor het resultaat van die wissel tot alle elementen in de goede volgorde staan.

Bijvoorbeeld: $\underline{32}1 \longrightarrow 2\underline{31} \longrightarrow \underline{21}3 \longrightarrow 123$. Een andere manier om hetzelfde resultaat te bereiken: $3\underline{21} \longrightarrow \underline{31}2 \longrightarrow 1\underline{32} \longrightarrow 123$. Wat ook de volgorde van de locale wissels is, het resultaat is uiteindelijk dezelfde gesorteerde lijst: het zoeken naar alternatieve oplossingen is hier dus niet zinvol.

Definieer een predicaat `xsort/2` dat het algoritme hierboven implementeert. De aanroep `xsort(List,Sorted)` slaagt als `Sorted` een gesorteerde permutatie is van `List`. Voor het gemak nemen we aan dat `List` een lijst van getallen is. Je kan `append/3` gebruiken om een opeenvolging van twee elementen die niet in de goede volgorde staan te zoeken en om ze om te wisselen. Gebruik `'!'` om ongewenst zoeken naar alternatieven af te breken.

1.2

Het begrip *subterm* is als volgt gedefinieerd: (i) een term is een subterm van zichzelf, (ii) een term t is een subterm van t' als t een subterm is van een argument van t'. In Prolog kan die definitie er zo uitzien:

```
subterm(Sub, Term) :- subsumes(Sub,Term).
subterm(Sub, Term) :-
    compound(Term),
    arg(_, Term, Arg),
    subterm(Sub, Arg).
```

Het ingebouwde `subsumes/2` is eenrichtingsunificatie: `subsumes(Algemeen,Specifiek)` slaagt als `Algemeen` gelijk gemaakt kan worden aan `Specifiek` zonder eventuele variabelen in `Specifiek` te instantiëren: `subsumes(X,a)` slaagt, maar `subsumes(a,X)` faalt.

Hieronder een voorbeeldaanroep.

```
?-subterm(Sub,f(a,g(X,b))).
Sub = f(a, g(X, b)); Sub = a; Sub = g(X, b); Sub = X; Sub = b; false.
```

Opdracht Definieer een variant `subterm/3`, waarbij `subterm(Term,Sub,Context)` slaagt als `Sub` een subterm is van `Term` en `Context` de context van `Sub` in `Term`. Een context is een term met een gat erin. Voor contexten gebruiken we de notatie `Var^Term` waarbij `Var` een Prolog variabele is, en `Term` een term die een voorkomen van die variabele bevat.

Hieronder een paar voorbeeldenaanroepen. Merk op dat `X^X` de lege context representeert (zoals `L-L` de lege lijst bij verschillijsten), en dat de aanroep tweemaal slaagt voor subterm `a`: er zijn immers twee voorkomens van `a` met verschillende contexten.

```
?- subterm(f(a,g(X,a)),Sub,Context).
Sub = f(a, g(X, a)),
Context = _G352^_G352 ;
Sub = a,
Context = _G352^f(_G352, g(X, a)) ;
Sub = g(X, a),
Context = _G352^f(a, _G352) ;
X = Sub,
Context = _G352^f(a, g(_G352, a)) ;
Sub = a,
Context = _G352^f(a, g(X, _G352)) ;
false.
```

Hint Gebruik in plaats van `arg/3` uit de definitie van `subterm/2` hierboven het ingebouwde predicaat `'=..'/2` om een complexe term te decomponeren tot een lijst `[Functor|Args]`. Bijvoorbeeld:

```
?- ?- f(a,g(X,a)) =.. [Functor|Args].
Functor = f,
Args = [a, g(X, a)].
```

Definieer een hulppredicaat `subterm_list` waarmee je `subterm/3` dan recursief kan aanroepen voor de elementen van de lijst van argumenten `Args`. Gebruik opnieuw `'=..'/2` om de contextterm op te bouwen, als je eenmaal weet welk argument je door de variabele voor het 'gat' moet vervangen.

1.3

Met behulp van `subterm/3` kunnen allerlei operaties op termen eenvoudig uitgedrukt worden in termen van unificatie, zoals we dat eerder zagen met concatenatie op basis van verschillijsten.

Definieer een predicaat `substitute/4` aan de hand van `subterm/3` van de vorige opgave. De aanroep `substitute(Oud,Nieuw,TermOud,TermNieuw)` slaagt als `TermNieuw` de term is die je krijgt als je een voorkomen van `Oud` in `TermOud` vervangt door `Nieuw`. Je mag ervan uitgaan dat `TermOud` een term is die geen vrije variabelen bevat (een zogenaamde *ground term*).

Bijvoorbeeld ("vervang een voorkomen van `a` in `f(a,g(a,d))` door `b(c,d)`"):

```
?- substitute(a,b(c,d),f(a,g(a,d)),Nieuw).
Nieuw = f(b(c, d), g(a, d)) ; Nieuw = f(a, g(b(c, d), d)) ; false.
```

Je definitie voor `substitute/4` vult simpelweg de argumenten van `subterm/3` op de goede manier in om unificatie zijn werk te laten doen:

```
substitute( ... ) :- subterm( ... ).
```

2 Metaprogrammeren

In de opdrachten van dit onderdeel verbinden we twee klassieke thema's uit de programmeerkunde met elkaar: binaire bomen, en rijtjes van welgeneste haakjes.[1]

Voor de bomen bekeken we het speciale geval van *gebalanceerde* binaire bomen. De `bintree/2` code hieronder is voor het algemene geval: `bintree(Boom,K)` slaagt als `Boom` een binaire boom is met `K` knopen (`K` in successornotatie).

```
/* bintree([],0).
   bintree(t(L,_,R),s(N)) :-
      add(X,Y,N),
      bintree(L,X),
      bintree(R,Y).

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

Het programma gebruikt `add/3`, de variant van `append/3` voor successorgetallen. De versie hieronder werkt met verschilgetallen, waardoor `add/3` wordt weggecompileerd. In Figuur 1 (achteraan) vind je de veertien binaire bomen met vier knopen.

[1] Zie Donald E. Knuth *The Art of Computer Programming*, Vol 4A, §7.2.1.6 'Generating all trees', Addison Wesley, 2011.

```
bintree(Tree,N) :- bintree(Tree,N,0).   % wrapper

bintree([],K,K).
bintree(t(L,_,R),s(K),K1) :-
    bintree(L,K,K0),
    bintree(R,K0,K1).
```

Nu de welgeneste rijtjes. Een rijtje $a_1 a_2 \ldots a_{2n}$ over het alfabet '(',')' is welgenest als het n voorkomens van haakje open en n voorkomens van haakje sluiten bevat, waarbij het k-de openingshaakje voorafgaat aan het k-de sluithaakje voor $1 \leq k \leq n$.

We kunnen op verschillende manieren een recursief recept geven voor welgeneste rijtjes: (i) het lege rijtje is welgenest; (ii) een rijtje is welgenest als het van de volgende vorm is: $(s)t$, waarbij s, t welgeneste rijtjes zijn. Een alternatief voor (ii) is (ii'): een rijtje is welgenest als het van de vorm $s(t)$ is, met s, t welgeneste rijtjes. Hieronder de twee opties in dcg formaat. Optie (ii) is compatibel met de Prolog zoekstrategie, optie (ii') is dat niet door de linksrecursie. Commentarieer één van de opties uit en overtuig je daarvan met een poging om de rijtjes voor $n = 4$ te genereren;

```
par --> [].
par --> "(",par,")",par. % optie (ii)
par --> par,"(",par,")". % optie (ii')
```

2.1 Opdrachten

Er is een mooie correspondentie tussen binaire bomen en welgeneste rijtjes: de code voor `par/2` geeft twee manieren weer om de knopen van een boom te bezoeken. Optie (ii) betekent dan het volgende: schrijf een haakje open bij de wortel, gevolgd door de haakjes uit de linkerdochter en een sluithaakje, gevolgd door de haakjes uit de rechterdochter van de wortelknoop. Optie (ii') bezoekt eerst de knopen van de linkerdochter, gevolgd door de haakjes uit de rechterdochter tussen haakjes.

Opdracht Breid de code voor `par/2` uit met een extra argument voor de binaire boom waardoor de beschreven correspondentie duidelijk wordt. Figuur 1 geeft voor optie (ii) de haakjes die je afleest uit de veertien binaire bomen met vier knopen. Die optie genereert de welgeneste rijtjes in omgekeerd alfabetische volgorde.

Opdracht Zoals gezegd werkt optie (ii') niet met de ingebouwde Prolog zoekstrategie. Daar kunnen we verandering in brengen door te zoeken met een drempelwaarde. De drempel is het aantal knopen van de binaire boom corresponderend met een rijtje—anders gezegd: het aantal recursie stappen.

Breid de code voor `term_expansion/2` van `'--->'/2` uit met een extra paar argumenten voor de verschilgetallen, zoals je die in `bintree/3` vindt. Genereer de veertien welgeneste

rijtjes van lengte $2n$ voor $n = 4$ aan de hand van optie (ii'). Wat kan je opmerken over de volgorde waarin de oplossingen worden opgeleverd? Hieronder alvast een wrapper (`d2s/2` zet decimale notatie om in successornotatie).

```
par(Drempel,Boom,Rijtje) :- d2s(Drempel,K),par(Boom,K,0,Rijtje,[]).
```

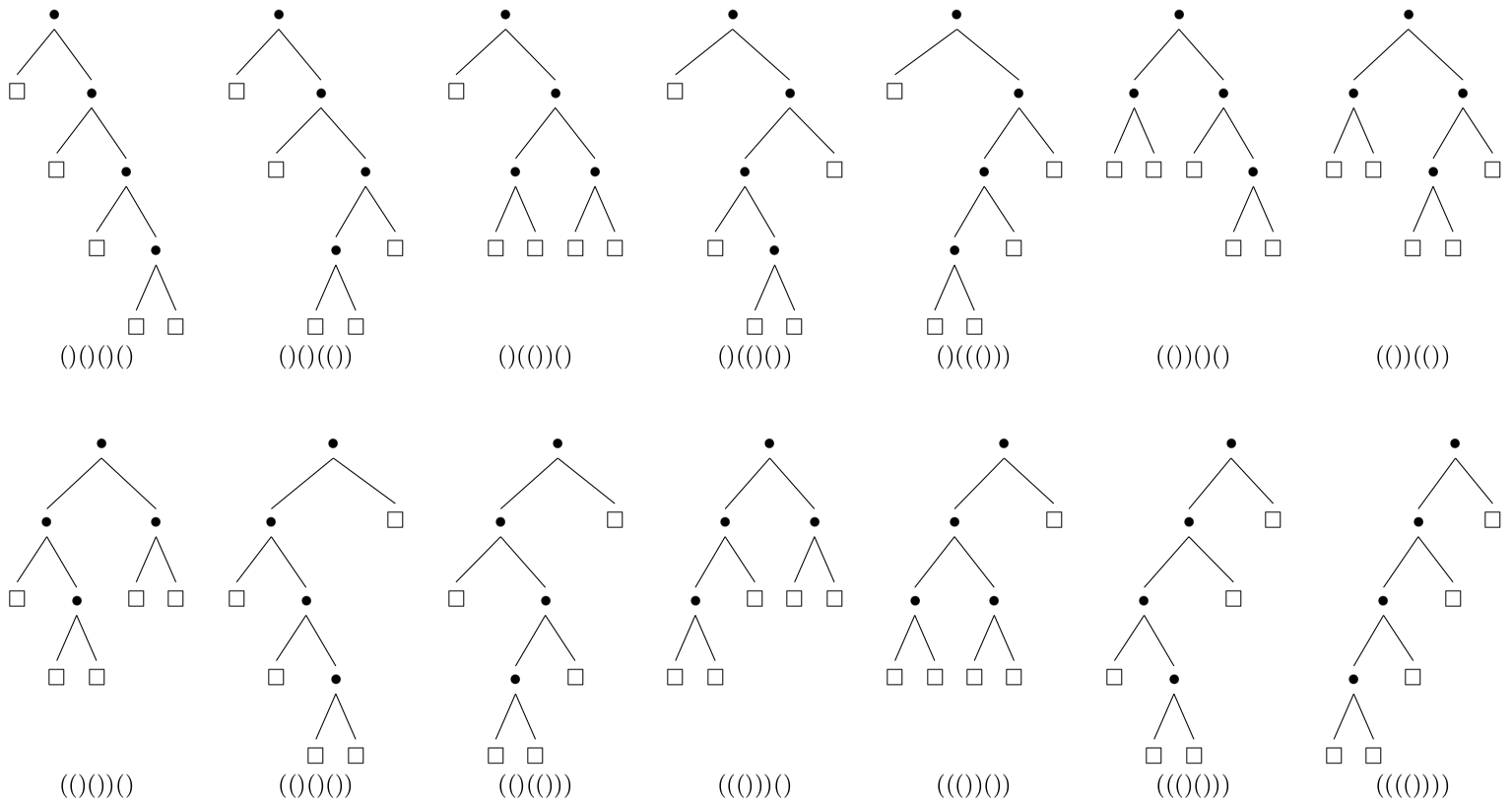
Opdracht De code van de vorige opdracht implementeert *bounded search*: recursie wordt afgebroken zodra een expliciet gegeven drempelwaarde is bereikt. Definieer een predicaat `par_consec/3` waarmee je *consecutively bounded* zoeken implementeert. De aanroep `par_consec(Drempel,Boom,Rijtje)` genereert binaire bomen `Boom` met bijhorend welgenest `Rijtje` voor *oplopende* drempelwaarden.

Voor het grensgeval gedraagt `par_consec/3` zich precies zoals `par/3`. Voor het recursieve geval hoog je de drempel op: als het niet meer lukt een oplossing te vinden voor drempelwaarde k, roep je `par_consec/3` aan met drempelwaarde $k + 1$.

Met een wrapper kan je tenslotte het expliciet vermelden van een drempelwaarde vermijden.

```
par_consec(Boom,Rijtje) :- par_consec(0,Boom,Rijtje).
```

□



Figuur 1: De 14 binaire bomen met 4 knopen en de bijhorende welgeneste rijtjes in omgekeerd alfabetische volgorde.