Lösningsförslag för Tenta i Programmeringsparadigm 2015-01-16 14.00-17.00

Funktionell Programmering

1 (a) Funktionell paradigm har troligen använts, eftersom rekursion används för upprepningen, värden skickas med anropet, och inga variabler används.

(b) En loop skulle användas för upprepningen och i varje steg uppdateras variabler.

(c)
```
f funk [1,2,3] []
    f funk [2,3] [funk 1]
    f funk [3] [funk 1, funk 2]
    f funk [] [funk 1, funk 2, funk 3]
[(funk 1), (funk 2), (funk 3)]
```

(d) Den svansrekursiva lösningen bygger inte på stacken i samma omfattning som den i uppgiften givna rekursiva lösningen.

(e) `f funk lista = [funk elem | elem <- lista]`

2 (a) `treeFnc :: (t -> a) -> Tree t -> Tree a`

(b) Givet en funktion (argument 1) och ett träd (argument 2), returnerar den ett nytt träd som fås från indataträdet genom att:

- (a) Byta plats på vänster och höger delträd i varje nod.
- (b) Applicera den givna funktionen på datat i varje nod.

(c) `Node (Node (Leaf 6) 5 (Leaf 4)) 3 (Leaf 2)`

Logikprogrammering

3 Här används indentering för att förtydliga anropsnivåerna.

(a)
- Anrop av `permutation([0], Y)`.
  - Första regeln matchar inte.
  - Andra regeln matchar, vilket resulterar i unifieringen `{E = 0, X = []}`.
    - Rekursivt anrop av `permutation([], Y1)`. Första regeln matchar med unifieringen `{Y = []}`, och vi returnerar.
    - Vi får unifieringen `{Y1 = []}`.
    - Anropet av `append(Y2, Y3, [])` ger unifieringen `{Y2 = [], Y3 = []}`.
    - Anropet av `append([], [0 | []], Y)` ger unifieringen `{Y = [0]}`, och vi returnerar.
- Vi får unifieringen `{Y = [0]}`.

(b)
- Anrop av `permutation([1,2], Y)`.
  - Första regeln matchar inte.
  - Andra regeln matchar, vilket resulterar i unifieringen `{E=1, X=[2]}`.
    - Rekursivt anrop av `permutation([2], Y1)`, resulterar i unifieringen `{Y1 = [2]}` enligt kontrollflödet i (a)-uppgiften.

  · Anropet av `append(Y2, Y3, [2])` ger unifieringen `{Y2=[], Y3=[2]}` (det är OK om man antar att den istället kommer ge `Y2=[2]` och `Y3=[]`).
  · Anropet av `append([], [1 | [2]], Y)` ger unifieringen `{Y = [1,2]}`, och vi returnerar.
- Vi får unifieringen `{Y = [1,2]}`.

4 Möjlig lösning:

```
count(X, [], 0).
count(X, [X|T], N) :-
   count(X, T, N1),
   N is N1+1, !.
count(X, [_|T], N) :-
   count(X, T, N).
```

5 (a) Programmet har den vanliga generera-och-testa strukturen:

```
tictactoe(X) :-
   board(X),  % generera
   legal(X).  % testa
```

där första predikatet `board(X)` används för att generera alla syntaktiskt korrekta tre-i-rad-bräden, dvs listor av längd 9 som innehåller konstanterna 'x', '0', eller '-', och det andra predikatet `legal(X)` används för att testa om ett bräde är giltigt enligt de givna reglerna.

(b) Predikate `board` skulle kunna se ut så här.

```
board([X11, X12, X13, X21, X22, X23, X31, X32, X33]) :-
   member(X11, [x, 0, -]),
   member(X12, [x, 0, -]),
   member(X13, [x, 0, -]),
   member(X21, [x, 0, -]),
   member(X22, [x, 0, -]),
   member(X23, [x, 0, -]),
   member(X31, [x, 0, -]),
   member(X32, [x, 0, -]),
   member(X33, [x, 0, -]).
```

(c) Predikatet `legal` skulle kunna se ut så här (använder sig av `count` från föregående uppgift):

```
legal(X) :-
   count(x, X, Nx),
   count(0, X, N0),
   Diff is Nx-N0,
   abs(Diff) <= 1, !.
```

Formella språk och syntaxanalys

6 (4 p)

(a) Kontextfria grammatiker är mer kraftfulla.

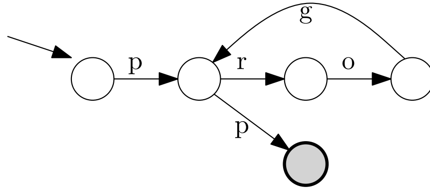
(b) Det finns oändligt många.

(c) En PDA är som en DFA men med ett (oändligt stort) minne i form av en stack.

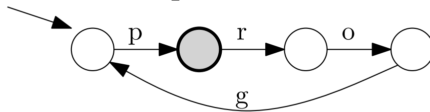
(d) Delsträngar (tokens) som en indatasträng är uppbyggd av.

7 (8 p)

(a) En DFA för L_1:



En DFA för L_2:



I båda lösningarna är tillståndet längst till vänster start-tillstånd, och det enda accepterande tillståndet är det gråmarkerade.

(b) $L_1 \cap L_2$ består bara av strängen "progp". Beskrivs av det reguljära uttrycket "`progp`".

(c) $L_1 \cup L_2$ består av alla strängar på formen "progprogprogprog...progp" ("prog" 0 eller flera gånger, följt av ett "p"), *och* alla strängar på formen "progrogrogrogrog...rogp" ("rog" 0 eller flera gånger, med "p" före och efter). Beskrivs av det reguljära uttrycket "`p(rog)*p|(prog)*p`".

(d) $L_1 \setminus (L_1 \cap L_2)$ består av alla strängar från L_1 förutom "progp". Detta är strängar på formen "progrogrog...rogp" där vi har "rog" antingen 0 gånger, eller minst 2 gånger (det får alltså inte förekomma exakt 1 gång). Några möjliga reguljära uttryck är:

```
pp|prog(rog)+p
p(p|rog(rog)+p)
p(rog(rog)+)?p
```

3.3 (8 p)

(a) En möjlig lösning. Icke-slutsymboler är Graph (startsymbol), NodeDesc, Neighbors, och NodeList.

$$\begin{aligned}
\text{Graph} &\rightarrow \text{NodeDesc} \mid \text{NodeDesc}\ \text{COMMA}\ \text{Graph} \\
\text{NodeDesc} &\rightarrow \text{LPAREN}\ \text{NAME}\ \text{COMMA}\ \text{LBRACE}\ \text{NodeList}\ \text{RBRACE}\ \text{RPAREN} \\
\text{Neighbors} &\rightarrow \epsilon \mid \text{NodeList} \\
\text{NodeList} &\rightarrow \text{NAME} \mid \text{NAME}\ \text{COMMA}\ \text{NodeList}
\end{aligned}$$

En annan möjlig lösning. Icke-slutsymboler är Graph (startsymbol), GraphTail, NodeDesc, Neighbors, och NodeListTail.

$$\begin{aligned}
\text{Graph} &\rightarrow \text{NodeDesc}\ \text{GRAPHTAIL} \\
\text{GraphTail} &\rightarrow \epsilon \mid \text{COMMA}\ \text{NodeDesc}\ \text{GraphTail} \\
\text{NodeDesc} &\rightarrow \text{LPAREN}\ \text{NAME}\ \text{COMMA}\ \text{LBRACE}\ \text{Neighbors}\ \text{RBRACE}\ \text{RPAREN} \\
\text{Neighbors} &\rightarrow \epsilon \mid \text{NAME}\ \text{NodeListTail} \\
\text{NodeListTail} &\rightarrow \epsilon \mid \text{COMMA}\ \text{NAME}\ \text{NodeListTail}
\end{aligned}$$

(b) Den första lösningen ovan är inte LL(1), eftersom t.ex. regeln för Graph har två möjligheter som båda börjar på NodeDesc. (Om man kastar om det andra alternativet för Graph så att det istället är "Graph COMMA NodeDesc" är grammatiken inte heller LL(1) eftersom grammatiken då blir vänster-rekursiv.)

Den andra lösningen är däremot LL(1). För varje produktionsregel kan man titta på nästföljande tecken och därigenom avgöra vilken av de olika möjligheterna som ska användas. (Detta är inte helt uppenbart, man måste titta på vad som kan komma efter ϵ-reglerna.)