Lösningsförslag för Tenta i Programmeringsparadigm 2015-04-10 08.00-11.00

Funktionell Programmering

1 ((a) Värdet 2 knyts till minnesutrymmet som kallas x, en anonym funktion `(\y -> 2*y)` returneras som resultat av anropet

(b) `lista :: [Double -> Double]`

(c) `[12.0,3.0,6.0]`

(d) `funk2 xs indata = [x indata|x <- xs]`

(e) Det används när `help` anropas med ett argument som t.ex. `(help 2)` som förklaras i deluppgift (a)

2 (a) Par av olika typ kan konstrueras med samma konstruktion (OBS! Parets delar är av samma typ). Det kallas polymorfi.

(b)
```
fram (EttPar a b) = a
bak  (EttPar a b) = b
```

(c) T.ex. `likaPar (EttPar 1 2) (EttPar 1 2)`

(d) Man kan åstadkomma detta genom att ändra typsignaturen till

```
likaPar :: Par Integer -> Par Integer -> Bool
```

Logikprogrammering

3
- Målet `member(5, L)` lyckas med första regeln och unifierar `H1 = 5` och `L = [5 | _G1]` för en obunden ny variabel `_G1`. Kontrollflödet fortsätter mot nästa mål `member(-3, L)`.
- Målet `member(-3, L)` matchar inte första regeln, men matchar andra regeln och unifierar `X1 = -3` och `T1 = _G1`. Kontrollflödet fortsätter med ett (rekursivt) anrop av `member(-3, _G1)`.
- Målet `member(-3, _G1)` lyckas med första regeln och unifierar `H2 = -3` och `_G1 = [-3 | _G2]` för en obunden ny variabel `_G2`. Målet `member(-3, L)` lyckas alltså med unifieringen `L = [5, -3 | _G2]`. Kontrollflödet fortsätter mot nästa målet `member(8, L)`.
- Målet `member(8, L)` matchar inte första regeln, men matchar andra regeln och unifierar `X2 = 8` och `T2 = [-3 | _G2]`. Kontrollflödet fortsätter med ett (rekursivt) anrop av `member(8, _G2)`.
- Målet `member(8, _G2)` lyckas med första regeln och unifierar `H3 = -3` och `_G2 = [8 | _G3]` för en obunden ny variabel `_G3`. Målet `member(8, L)` lyckas alltså med unifieringen `L = [5, -3, 8 | _G3]`.

I slutet får vi alltså svaret `L = [5, -3, 8 | _G3]`.

4 Ja, det finns en viss konflikt. Betrakta till exempel frågan `member(2, [2, 3, 4])`. Den lyckas vid första regeln (svar: true), och detta borde vara det enda svaret till frågan. Men det är det inte: det finns ett svar till, nämligen false.

Man kan rätta till detta och lägga ett snitt i slutet av första regeln:

```
memberCut(H, [H | _]) :- !.
memberCut(X, [_ | T]) :- memberCut(X, T).
```

Frågan `memberCut(2, [2, 3, 4])` ger nu bara ett svar, true. Men om vi skulle vilja använda predikatet för att generera (enligt generera-och-testa programmeringstekniken) alla dess element med målet `memberCut(X, [2, 3, 4])`, kommer vi bara få ut första svaret `X=2`.

5 (a) `T ::= leaf(D) | lbranch (D, T) | rbranch(D, T) | branch (D, T, T)`

(b) En möjlig lösning:

```
insel(X, leaf(D), lbranch(D, leaf(X))) :- X<D.
insel(X, leaf(D), rbranch(D, leaf(X))).

insel(X, lbranch(D, T), lbranch(D, TX)) :- X<D, insel(X, T, TX).
insel(X, lbranch(D, T), branch(D, T, leaf(X))).

insel(X, rbranch(D, T), branch(D, leaf(X), T)) :- X<D.
insel(X, rbranch(D, T), rbranch(D, TX)) :- insel(X, T, TX).

insel(X, branch(D, T1, T2), branch(D, T1X, T2)) :- X<D, insel(X, T1, T1X).
insel(X, branch(D, T1, T2), branch(D, T1, T2X)) :- insel(X, T2, T2X).
```

(c) Svar (för lösningen ovan, andra lösningar kan ge andra svar):

```
T1 = rbranch(3, leaf(5)),
T2 = branch(3, leaf(-2), leaf(5)),
T3 = branch(3, leaf(-2), lbranch(5, leaf(3))) .
```

Formella språk och syntaxanalys

6 (5 p)

(a) En sträng är en sekvens/lista med tecken. Ett språk är en mängd av strängar. En språkklass är en mängd av språk.

(b) Rekursiv medåkning är en algoritm/metod för att parsa kontextfria grammatiker (den kan dock inte hantera alla kontextfria grammatiker). I rekursiv medåkning har man en funktion/metod för varje icke-slutsymbol i grammatiken, och denna funktion tittar på nästa indatasymbol för att avgöra vilken produktionsregel som ska användas och anropar rekursivt funktionerna för ev. icke-slutsymboler som i sin tur behöver parsas.

7 (4 p)

(a) Falskt, eftersom det inte finns någon riktad cykel i automaten.

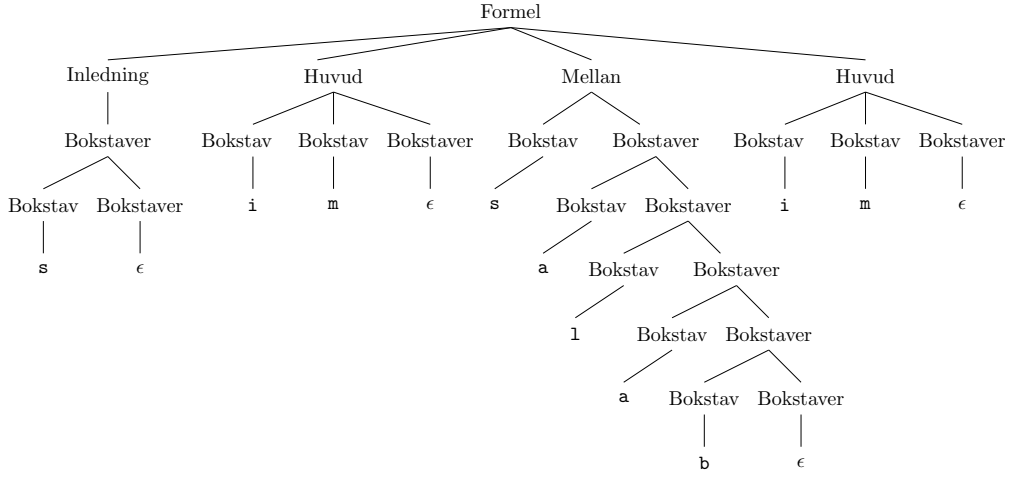
(b) Sant. Detta är sant oavsett hur DFA:n ser ut, för varje DFA finns ett reguljärt uttryck som beskriver samma språk.

8 (11 p)

(a) Grammatiken är inte korrekt. Grammatiken tvingar inte de två huvuddelarna att vara samma sträng, olika förekomster av "Huvud" kan vara olika strängar.

(b) Grammatiken är tvetydig. T.ex. kan strängen "`simsalabim`" tolkas som "Inledning = `s`, Huvud1 = `im`, Mellan = `salab`, Huvud2 = `im`", eller "Inledning = `sim`, Huvud1 = `sala`, Mellan = `b`, Huvud2 = `im`" (samt många fler varianter).

(c) En möjlig lösning visas här. Eftersom grammatiken är väldigt tvetydig finns många andra lösningar (jfr svaret på föregående deluppgift).



(d) Ja, det är reguljärt.

Vi börjar med att bevisa påståendet i ledtråden för uppgiften. Givet en trollformel representerad som $xyzy$ där x är en sträng (inledande del), y är en sträng av längd mer än 2 (huvuddel), och z är en sträng (mellandel). Låt $y = w\tilde{y}$, där $\tilde{y}$ är de två sista tecknen i y och w är de $\text{length}(y) - 2$ första. Vi kan då representera samma trollformel som $(xw)\tilde{y}(zw)\tilde{y}$, dvs vi kan låta inledande delen vara xw, mellandelen vara zw, och huvuddelen $\tilde{y}$. Kontentan är alltså att om en sträng är en trollformel så kan den representeras så att huvuddelen har längd exakt 2, och vi kan därför kräva att huvuddelen ska ha längd 2 utan att ändra vad som är eller inte är en giltig trollformel.

Språket av trollformler där huvuddelen har längd exakt 2 är reguljärt eftersom det bara finns ändligt många olika möjliga huvuddelar. För trollformler där huvuddelen t.ex. är "aa" har vi det reguljära uttrycket ".*aa.+aa". När huvuddelen är "ab" har vi ".*ab.+ab". Ett reguljärt uttryck för trollformler ges då av att ta alterneringen av alla dessa uttryck, dvs.

```
.*aa.+aa|.*ab.+ab|.*ac.+ac| ...
```

(Principen är densamma som i uppgift 2 i labb S1.)