Lösningsförslag för Tenta i Programmeringsparadigm 2016-01-15 14.00-17.00

Funktionell Programmering

1 (a)
```
char2bin "Gröt"
char2bin "röt"
char2bin "öt"
'ö' : (char2bin "t")
'ö' : (char2bin "")
'ö' : []
"ö"
```

(b) I de anrop som görs av funktionen `char2bin` skickas uppdaterade värden som parametrar till rekursiva anrop av funktionen. Se även beskrivningen av beräkningsstegen.

(c) Funktionen är inte svansrekursiv eftersom operatorn `:` använder resultatet av det rekursiva anropet som indata. Det rekursiva anropet är alltså inte det sista som sker i funktionen.

(d) Möjlig lösning:

```
char2binS text = char2binSv text []

char2binSv [] resultat = resultat
char2binSv (t:text) resultat
   | svTecken t = char2binSv text (resultat++[t])
   | otherwise = char2binSv text resultat
```

(e) Möjlig lösning:

```
char2binA text = [ts |ts <- text, svTecken ts]
```

2 (a)
```
ys :: [Float -> Float]
```

(b) En anonym funktion som tar ett argument av typen Float som indata. Funktionen adderar indata med ett.

3 (a) `(+)` görs till en överlagrad operator för add

(b)
```
(+) (Vector2 1 2) (Vector2 3 4)
```

Logikprogrammering

4 (8 p)

1. Mål: `member(2, L).`
2. Skapar instans av första regeln: `member(X1, [X1|A1]).`
   Lyckas med unifieringen: `X1=2, L=[2|A1]`.
   Går vidare till nästa mål.
3. Mål: `member([2|A1], [[1, 2]]).`
4. Skapar instans av första regeln: `member(X2, [X2|A2]).`
   Lyckas inte unifiera, ty listan `X2` inte kan börja med `2` och med `1` samtidigt.
   Går vidare till nästa regel.

5. Skapar instans av andra regeln: `member(X3, [A3|T1]) :- member(X3, T1).` Unifierar: `X3=[2|A1], T1=[].`
   5.1. Mål: `member([2|A1], []).`
   5.2. Matchar ingen regel, backtrackar till 2.
2. Skapar instans av andra regeln: `member(X4, [A4|T2]) :- member(X4, T2).` Unifierar: `X4=2, L=[A4|T2].`
   2.1. Mål: `member(2, T2).`
   2.2. Skapar instans av första regeln: `member(X5, [X5|A5]).` Lyckas med unifieringen: `X5=2, T2=[2|A5].`

   Går vidare till nästa mål.
3. Mål: `member([A4|[2|A5]], [[1, 2]]).`
4. Skapar instans av första regeln: `member(X6, [X6|A6]).` Lyckas med unifieringen: `X6=[1, 2], A4=1, A5=[], A6=[].`

Svar: `L=[1, 2]`. (Egentligen: `L=[1|[2|[]]]`)

5 (2 p) Antagandet om en sluten värld innebär att allt som är sant är känt i systemet, och därmed betraktas allt som man inte vet att vara sant som falskt. I Prolog-sammanhang betyder detta, att om utvärderingen av ett mål terminerar men inte lyckas, då betraktas målet som falskt. Till exempel kommer frågan `?- happy(nisse).` returnera `false` i följande lilla kunskapsbas:

```
happy(anna).
happy(karl).
```

Negation returnerar det negerade värdet gentemot svaret enligt ovan. Frågan `?- \+ happy(nisse).` i kunskapsbasen ovan kommer alltså returnera `true`. (Negation betyder därmed inte logisk negation, utan negerar bevisbarheten av ett mål.)

6 (10 p)

(a) Bristen med med denna implementation är att den kan leda till icke-terminerande beräkningar. Problemet är att villkoret att inte ha upprepningar av städer kollas på efterhand, efter en ny stig ska genereras. Istället måste villkoret byggas i själva genereringen av stigarna. En samling fakta vore:

```
edge(a, b).
edge(b, a).
edge(b, c).
edge(c, d).
```

På frågan `?- route(a, d, R).` kommer koden gå in i en oändlig loop.

(b) Ett sätt att lösa problemet är att införa en extra parameter `NotVia` som är en lista över städer som inte får passeras. I början är listan tom, och fylls gradvis med städerna som redan har passerats.

```
route(From, To, Route) :-
   routeConstr(From, To, Route, []).

routeConstr(From, To, [From, To], NotVia) :-
   edge(From, To).
routeConstr(From, To, [From, Next | Route], NotVia) :-
   edge(From, Next),
   \+ member(Next, NotVia),
   routeConstr(Next, To, [Next | Route], [From | NotVia]).
```

Formella språk och syntaxanalys

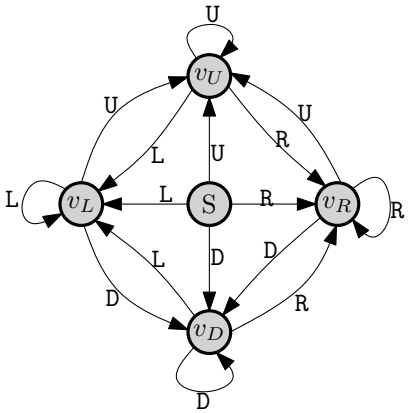
7 (12 p)

(a) Ja, språket är reguljärt.

Två möjliga motiveringar:

Motivering 1. Det enda vi behöver minnas är vilken av de fyra riktningarna det föregående steget som togs var. Språk som bara kräver konstant mängd minne kan kännas igen av ändliga automater (vi kan ha ett tillstånd för varje möjlig minneskonfiguration) och språket är alltså reguljärt.

Motivering 2. Språket känns igen av följande automat med starttillstånd S.



Alla tillstånden är accepterande. Här betyder t.ex. tillstånd v_L att det senaste steget man tog var ett steg åt vänster. (Detta är den automat man får om man tillämpar det mer abstrakta argumentet i den första motiveringen.)

(b) Nej, språket är inte reguljärt.

Rent intuitivt så behöver vi kunna räkna exakt hur många steg vi gått åt varje håll, och eftersom vi kan ta godtyckligt många steg kräver detta *obegränsat* minne, men en automat har bara begränsat minne.

Ett lite mer rigoröst argument är följande. Låt L_1 vara språket vi är intresserade av, och L_2 vara alla strängar på formen $\texttt{L}^n\texttt{UR}^m\texttt{D}$ (vägbeskrivningar som tar n steg åt vänster, ett steg uppåt, m steg åt höger, och sedan ett steg nedåt. Språket L_2 är reguljärt, eftersom man kan beskriva det med uttrycket `L*UR*D`. Språket $L_1 \cap L_2$ består då av alla strängar på formen $\texttt{L}^n\texttt{UR}^m\texttt{D}$ där $n < m$. Detta är inte reguljärt (man kan visa det med det s.k. pumping-lemmat, eller reducera det till språket 0^n1^n, eller nöja sig med att notera att det här är tydligt att man behöver kunna räkna för att jämföra n och m). Det betyder att L_1 inte heller är reguljärt (om L_1 vore reguljärt skulle $L_1 \cap L_2$ också vara reguljärt eftersom vi vet att L_2 är reguljärt).

(c) Ja, språket är reguljärt.

Man kan motivera det på samma sätt som motivering 1 i (a)-uppgiften: det enda vi behöver minnas är exakt vilken position i labyrinten vi befinner oss på. Eftersom det finns ett fixerat ändligt antal positioner går detta att göra med än ändlig automat.

Mer konkret kan vi alltså konstruera en automat där varje tillstånd är en av positionerna i labyrinten, med övergång från position x till position y på tecken $c \in \{L, R, U, D\}$ om instruktionen c tar oss från position x till position y.

(d) Ja, språket är reguljärt.

Eftersom det bara finns ett ändligt antal positioner i labyrinten kan vägbeskrivningarna bara vara ändligt långa, vilket i sin tur betyder att språket är ändligt, och alla ändliga språk är reguljära.

8 (8 p)

(a) Exempel som tillhör språket (bara tre behövde anges, vi listar några fler): `cp, cz, cfp, cfz, cfbp, ccpcp, ccfzcfz, dp, dz, dcp, ddp, ddddddddp`. Exempel som inte tillhör språket (bara ett behövde anges): `ccp, dcpcp` och alla strängar som innehåller andra tecken än `bcdfpz`.

(b) En rekursiv medåknings-parser för språket kommer att han en funktion för var och en av de tre icke-slutsymbolerna Start, Cat, och Dog.

Pseudo-kod för detta visas nedan. Den använder sig av hjälpfunktioner PeekToken() (som kollar vilket tecken som finns på aktuell position i indata) och NextToken() (som stegar fram ett steg i indata).

Algorithm 1: Start()

```
1  begin
2      x ← PeekToken()
3      if x = 'c' then
4          NextToken()
5          Cat()
6      else if x = 'd' then
7          NextToken()
8          Dog()
9      else
10         SyntaxError()
```

Algorithm 2: Cat()

```
1  begin
2      x ← PeekToken()
3      if x = 'p' then
4          NextToken()
5      else if x = 'z' then
6          NextToken()
7      else if x = 'f' then
8          NextToken()
9          Dog()
10     else
11         Start()
12         Start()
```

Algorithm 3: Dog()

```
1  begin
2      x ← PeekToken()
3      if x = 'p' then
4          NextToken()
5      else if x = 'z' then
6          NextToken()
7      else if x = 'b' then
8          NextToken()
9          Cat()
10     else
11         Start()
```

Funktionen Cat() är t.ex. ansvarig för att parsa ett element av typen Cat. Den kollar på nästa tecken i indata, och avgör baserat på detta vilken av de fyra produktionsreglerna för Cat som ska användas (genom If-satserna med fyra olika fall).

(Den givna pseudokoden kontrollerar bara om indata tillhör språket. I praktiken vill man antagligen även konstruera och returnera ett syntaxträd medan man parsar.)

(c) Nej, grammatiken är inte tvetydig.

Via uppgift (b) ser vi att grammatiken kan parsas med rekursiv medåkning (dvs är en $LL(1)$-grammatik). Sådana grammatiker är inte tvetydiga. (Om man inte löser (b) kan man fortfarande argumentera på annat sätt för att grammatiken inte är tvetydig.)
