Lösningsförslag för Tenta i Programmeringsparadigm 2017-04-11 08.00-11.00

Funktionell Programmering

1. (a) Ta in en int och returnera en sträng med dess binära representation.

```
toBinary::Int->String
toBinary n
    |n==0 = "0"
    |otherwise = toBinaryHelper n ""

toBinaryHelper::Int->String->String
toBinaryHelper n answer
    |n == 0 = answer
    |n `mod` 2 == 1 = toBinaryHelper (n `div` 2) ('1':answer)
    |otherwise      = toBinaryHelper (n `div` 2) ('0':answer)
```

Alternativ lösning:

```
toBinary n =
    | n <= 1    = show(n)
    | otherwise = toBinary(n `div` 2) ++ show(n `mod` 2)
```

Krånglig men korrekt lösning:

```
import Data.Bits

toBinary::Int->String
toBinary n =
    toBinaryHelper n 1 ""

toBinaryHelper::Int->Int->String->String
toBinaryHelper n mask answer
    |mask>n = answer
    |(.&.) mask n == mask = toBinaryHelper n (mask `shiftL` 1) ('1':answer)
    |otherwise = toBinaryHelper n (mask `shiftL` 1) ('0':answer)
```

   (b) Denna lösning använder det första lösningsförslaget ovanför:

```
toBinary 21
toBinaryHelper 21 ""
toBinaryHelper 10 "1"
toBinaryHelper 5 "01"
toBinaryHelper 2 "101"
toBinaryHelper 1 "0101"
toBinaryHelper 0 "10101"
```

2. (a) `[1,4,9,16]`

   (b)
```
listCreator2::Int->Int->Int->[Int]
listCreator2 limit power n =
    [x^power|x<-[1..n], x^power < limit]
```

3.
   - Alternativ 1: Parametrisk polymorfi ger 2p.
   - Alternativ 2: Typvariabel ger 2p.
   - Alternativ 3: Polymorfi ger 1p.

4. Summan av de första n udda talen.

```
firstNOdd::Int -> Int
firstNOdd n =
    foldr (\a b->a+b) 0 [1,3..2*n]
```

   Rättningsmall:

   - kommatecken mellan a och b i lambdanotationen -1p
   - Avsaknad av 0:an: -1p
   - Alla udda tal under n istället för de första n udda talen: -2p
   - Använder sum eller rekursion: totalt 0p

Logikprogrammering

4 1. Mål: `candVotes(eva, [adam, eva], X).`

1.1. Skapar instans av första regeln: `candVotes(C1, [], 0).`
Kan inte unifiera `[adam, eva]` med `[]`.
Går till nästa regel.

1.2. Skapar instans av andra regeln:

```
candVotes(C1, [V1 | Vs1], N1) :-
    candVotes(C1, Vs1, M1), C1 = V1, N1 is M1+1.
```

Unifierar: `C1=eva, V1=adam, Vs1=[eva], N1=X`.
Går vidare till första delmålet.

1.3. Delmål: `candVotes(eva, [eva], M1).`

1.3.1. Skapar instans av första regeln: `candVotes(C2, [], 0).`
Kan inte unifiera `[eva]` med `[]`.
Går till nästa regel.

1.3.2. Skapar instans av andra regeln:

```
candVotes(C2, [V2 | Vs2], N2) :-
    candVotes(C2, Vs2, M2), C2 = V2, N2 is M2+1.
```

Unifierar: `C2=eva, V2=eva, Vs2=[], N2=M1.`
Går till första delmålet.

1.3.3. Delmål: `candVotes(eva, [], M2).`

1.3.3.1. Skapar instans av första regeln: `candVotes(C3, [], 0).`
Unifierar: `C3=eva, M2=0`.
Delmål 1.3.3 klart.

1.3.4. Nästa delmål: `C2 = V2`.

1.3.5. Nästa delmål: `N2 is M2+1`.
Utvärderar och unifierar `N2=1`.
Delmål 1.3 klart.

1.4. Nästa delmål: `C1 = V1`
Unifiering misslyckas (eftersom vi redan har bindningarna `C1=eva`, `V1=adam`).
Finns inga steg att backtracka till, sökningen misslyckas.

Svar: `no` / false.

5 Strukturen för en allmän generera-testa lösning är

```
solve(Problem, Solution) :-
    generateCandidate(Problem, Solution),
    testSolution(Solution).
```

6 (a) En möjlig representation:

- Noder är atomer `a`, `b`, `c`, ...
- Bågar är termer `edge(a, b)` där `a` och `b` är noder.
- Grafer är termer `graph(Nodes, Edges)` där `Nodes` är en lista av noder, och `Edges` är en lista av bågar.
- Vägar är listor av noder.

(b) I representationen ovan:

```
graph([a, b, c, d],
      [edge(a, b), edge(a, c), edge(b, a), edge(b, d),
       edge(c, b), edge(d, a), edge(d, c)])
```

(c)
```
hamiltonian(graph(Nodes, Edges), Path) :-
    permutation(Nodes, Path), % Generera en nodsekvens som innehåller
                              % alla noder exakt en gång.
    path(Path, Edges).        % Testa om den är en väg i grafen.

path([_], Edges) :- !.
path([N1, N2 | Ns], Edges) :-
    member(edge(N1, N2), Edges), !,
    path([N2 | Ns], Edges).
```

Formella språk och syntaxanalys

7 Möjlig lösning 1:

```
<Term> ::= <Sign> <First> <Rest>
<Sign> ::= '+' | '-'
<First> ::= <Int> | <VarPot>
<Rest> ::= '' | '*' <VarPot> <Rest>
<VarPot> ::= <Var> <Power>
<Var> ::= 'a' | 'b' | ... | 'z'
<Power> ::= '' | '^' <Int>
<Int> ::= <Digit> <IntRest>
<IntRest> ::= '' | <Digit> <IntRest>
<Digit> ::= '0' | '1' | ... | '9'
```

En annan variant (obs denna kan ej parsas med rekursiv medåkning, relevant för nästa uppgift!):

```
<Term> ::= <Sign> <Factors>
<Sign> ::= '+' | '-'
<Factors> ::= <Int> <Rest> | <VarPot> <Rest>
<Rest> ::= '' | '*' <VarPot> <Rest>
<VarPot> ::= <Var> | <Var> '^' <Int>
<Var> ::= 'a' | 'b' | ... | 'z'
<Int> ::= <Digit> | <Digit> <Int>
<Digit> ::= '0' | '1' | ... | '9'
```

8 I en rekursiv medåknings-parser har vi en funktion för varje icke-slutsymbol i vår grammatik, "ansvarig" för att parsa respektive icke-slutsymbol. Varje sådan funktion får titta på nästa tecken (eller token, om man gjort lexikal analys innan) i indata och baserat på detta besluta vilken produktionsregel som ska användas.

Pseudokod för lösningsförslag 1 ovan följer nu. Vi tänker oss att det finns två funktioner `peek()` och `next()` där det första kikar på vad nästa tecken i indata är (utan att konsumera det), och det andra stegar fram ett tecken i indata.

```
function Term():
  Sign()
  First()
  Rest()

function Sign():
  t = next()
  if t not in ['+', '-']: SyntaxError!

function First():
  t = peek()
  if isDigit(t):
     Int()
  else:
     VarPot()

function Rest():
  t = peek()
  if t == '*':
    next() % eat the crunchy asterisk
    VarPot()
    Rest()

function VarPot():
  Var()
  Power()

function Power():
  t = peek()
  if t == '^':
    next() % carets, yum!
    Int()

...etc
```

För lösningsförslag 2 på grammatiken ovan stöter vi på problem, då grammatiken inte är LL(1). Följande svar skulle därför inte ge full poäng men ge 5 poäng om svaret tydligt pekar ut vilka delar av grammatiken som är problematiska.

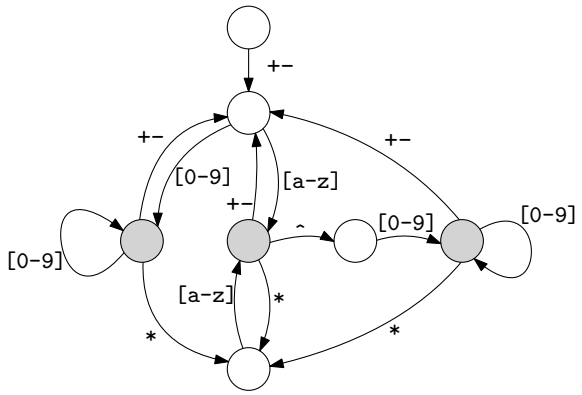
```
function Term():
  Sign()
  Factors()

[de flesta funktionerna ungefär som i första lösningsförslaget...]

function VarPot():
  t = peekToken()
  % Problem!! Här kan vi inte veta baserat på t vilken av de två
  % reglerna som ska väljas, vi behöver gå förbi första variabeln och
  % kolla på nästa token för att se om det kom en potens-del eller
  % inte.
  next()
  t = peekToken() % fuska lite och kolla ett tecken till innan vi bestämmer oss
  if t == '^':
    next()
    Int()

Vi får sedan samma problem i Int()-funktionen.
```

9 Angrepps-sätt: det enklaste är att stegvis bygga upp ett reguljärt uttryck från reglerna för de olika delarna. T.ex. kan vi skriva ett reguljärt uttryck för "variabel-potens" som "`[a-z](\^[0-9]+)?`".

Möjlig slutlösning:

```
([+-]([0-9]+|[a-z](^[0-9]+)?)(\*[a-z](^[0-9]+)?)*)+
```

Möjlig DFA:



10 (a) **Fortfarande reguljärt**.
Eftersom det bara finns ett ändligt antal (26) olika variabler, så när vi läser en term behöver vi "bara" komma ihåg vilken av alla 2^{26} olika delmängder av variabler som hittills använts i den aktuella termen.

(b) **Ej reguljärt.**
Det finns oändligt många olika variabel-potenser, t.ex. `x^1`, `x^2`, `x^3`, `x^4`, och en DFA kan inte komma ihåg exakt vilka av dessa som hittills observerats.