Övning 5 - Tillämpad datalogi 2013

```
0 # coding: latin
```

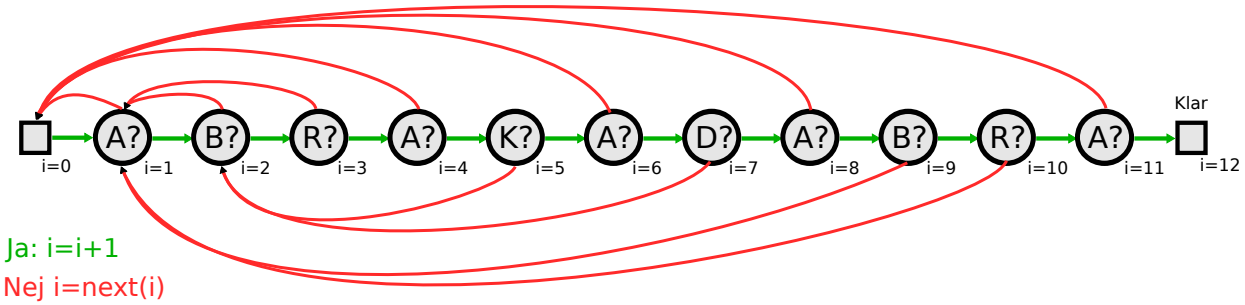
Automater, reguljära uttryck, syntax

Sammanfattning

Idag gick vi igenom **automater**, **reguljära uttryck** och **syntax**. Vi exemplifierade detta genom att lösa 5 uppgifter. Vi tittade på hur next vecktorn i en KPM automat byggs up, hur reguljära uttryck kan skapas och tolkas samt vi felsökte 4 olika syntax grammatiker och skapade får egna grammatik för att kolla om en websida är rätt skriven.

1. Abrakadabra

Konstruera en KMP-automat som söker efter texten ABRAKADABRA. Ange även den next-vektor som definierar automaten. ***Ungefär hur många jämförelser behövs för att automaten ska se att ordet inte finns med i "Harry Potter och Fenixorden", en bok på 1.8 MB?***

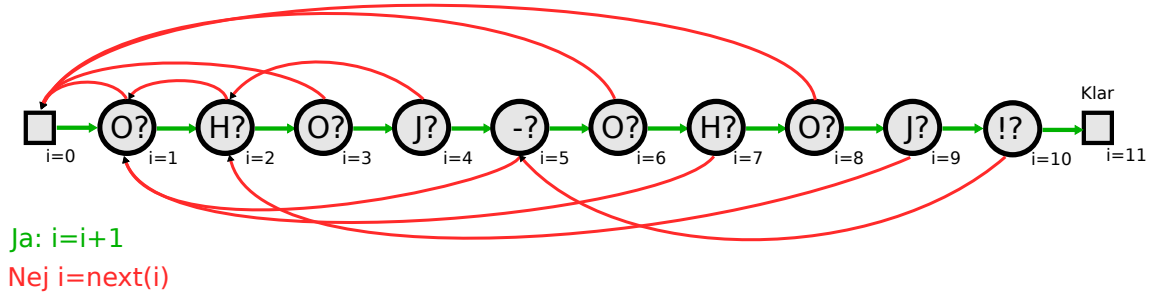


i	1	2	3	4	5	6	7	8	9	10	11
$next(i)$	0	1	1	0	2	0	2	0	1	1	0

KMP-sökning tar m + n jämförelser (bygga upp next vektorn och sen gå igenom), där m är antal tecken i söksträngen och n är antal tecken i texten. ASCII kod krväver en byte, vanligtvis 8 bits där en bit kan ta värdet 0 eller 1).
För varje tecken så vi får alltså 1.8 miljoner + 11 (alt. med bas två definitionen av en megabyte som då är $2^{20} = 1048576$ bytes så att vi får $1.8 * 2^{20} + 11 = 1887437 + 11$ jämförelser.

2. Värstingvrålsautomat (Tildatenta 000831)

Skriv en KMP-automat som söker efter OHOJ-OHOJ! i en texfil med nedtecknade värstingvrål från en vitmaktskonsert. Ange också den next-vektor som definerar automaten.



i	1	2	3	4	5	6	7	8	9	10
$next(i)$	0	1	0	2	1	0	1	0	2	5

Regular expression

Metatecken	Beskrivning
.	matchar *ett tecken*, vilket som helst (utom radslut)
X?	matchar noll eller X
X*	matchar noll eller flera X
X+	matchar en eller flera X
[ ]	matchar ett av tecknen inom klammrarna
X|Y	matchar uttrycket X eller uttrycket Y
\.	matchar en punkt. ("escape:a metatecken")
( )	skapar en grupp, tex (ab)+ ett eller flera ab:n
\n	refererar n:te gruppen (0-9)

For exampel, `ca*t` will match *ct* (0 a characters), *cat* (1 a), *caaat* (3 a characters), and so forth.

A step-by-step example will make this more obvious. Let's consider the expression `a[bcd]*b`. This matches the letter 'a', zero or more letters from the class [bcd] (b or c or d), and finally ends with a 'b'. Now imagine matching this RE against the string abcbd.

Step	Matched	Explanation
1	*a*	The *a* in RE matches
2	*abcbd*	The engine matches *[bcd]**, going as far as it can but the current position is to the end of the string.
3	*Failure*	The engine tries to match *b*, but the current position is at the end of the string to it fauls
4	*abcb*	Back up so that *[bcd]* matches one less character*
5	*Failure*	Try *b* again, but the current postion is at the last character, which is *'d'*
6	*abc*	Back up again, so that *[acd]** only mathces *bc*
7	abcb	Try *b* again. This time the character at the current position is *'b'*, so it succeeds.

$$http://docs.python.org/howto/regex.html regex-howto$$

```
110
111 import re
112 pattern="a[bcd]*b"
113 sometext="abcbd"
114 print re.findall(pattern, sometext)
```

3. Regexp

a) Givet det reguljära uttrycket

```
s(a|o)nd-?låda
```

Skriv upp tre strängar som matchas av det reguljära uttrycket och ett som inte gör det.

b) Söka efter Kronskog. Skriv ett reguljärt uttryck som matchar alla tänkbara -bara sätt att stava namnet Kronskog (Crounskog, Krohnskoog, etc).

1. sandlåda, sand-låda, sondlåda men inte syndlåda

2.
i) *'C'* eller *'K'* ⇒ `[CK]`
ii) alltid *ro* ⇒ `ro`
ii) eventuellt flöljt av ett *u* ⇒ `u?`
iv) eventuellt flöljt av ett *h* ⇒ `h?`
v) sen alltid *'nsko'* ⇒ `nsko`
vi) eventuellt flöljt av ett *'o'* ⇒ `o?`
vii) och alltid *'g'* på slutet ⇒ `h?`

Det ger oss det regulgära uttrycket `[CK]rou?h?nskoo?g` Observera att `[CK]ro[uh]?nskoo?g` (⇒ Kouhuhuhuhnskog), dvs multipla uh

4. Syntax för kanadensare (Tildatenta 040313)

Olle sitter och rättar ett tentatal. Tentatalet går ut på att man ska skriva en grammatik för meddelanden av följande typ:

Kanot 42, kanot 666, kanot 4711 och kanot 17 ska in!
Kanot 1 och kanot 2 ska in!
Kanot 13 ska in!

Vilken eller vilka av följande fyra alternativ kan producera dessa meddelan- den? Motivera med exempel varför de övriga inte kan producera dem.
En del av alternativen kan producera oönskade meningar, man vill t ex inte ha Kanot 1 och kanot 2, kanot 3 och kanot 4 ska in!
Vilket eller vilka av alternativen kan producera oönskade meningar? Ge exempel.

```
(1)  <meddelande> ::= Kanot <tal><svans> |
                      <meddelande> kanot <tal><svans>
     <svans> ::= och| ska in!  | ,
     <tal> ::= 1 | 2 | 3 | ...

(2)  <meddelande> ::= Kanot <tal> ska in!  |
                      Kanot <tal><svans>
     <svans> ::= och kanot <tal> ska in!  | ,
                 kanot <tal><svans>
     <tal> ::= 1 | 2 | 3 | ...

(3)  <meddelande> ::= Kanot <tal><svans>
     <svans> ::= ska in!  | ,
                 kanot <tal> |
                 och kanot <tal>
     <tal> ::= <svans> | 1 | 2 | 3 | ...

(4)  <meddelande> ::= Kanot <tal><svans> |
                      kanot <tal><svans>
     <svans> ::= ska in!  | , | och
     <tal> ::= 1 | 2 | 3 | ...
```

Observera att grammatik 1,2 och 3 har rekursiva anrop medans gramatik 4 inte har det. Detta borde leda till begränsningar för vad grammatik 4 kan kan göra.

Alternativ 1 och 2 kan producera alla meningarna.

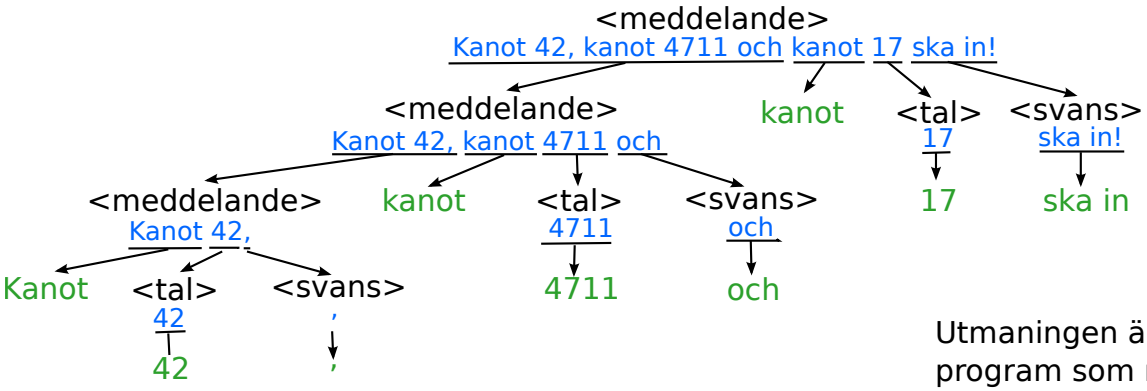
Alternativ 3 och 4 kan inte producera
Kanot 1 och kanot 2 ska in!

Alternativ 1 och 4 godkänner felaktigt
Kanot 4 och

Alternativ 3 godkänner felaktigt
Kanot och kanot 2

Exempel på syntaxträd:

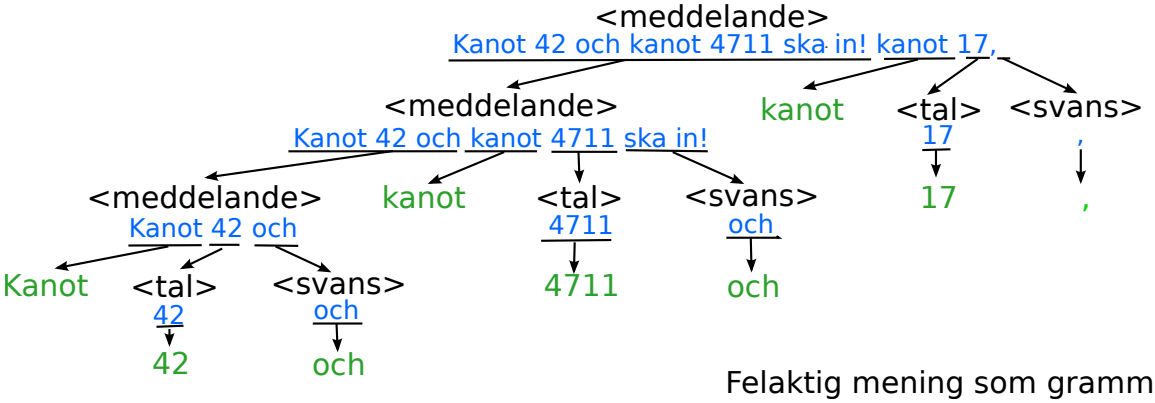
Mening at parsa: Kanot 42, kanot 4711 och kanot 17 ska in!



Utmaningen är att skriva ett program som **rekursivt** kan bygga trädet och identifiera att varje löv är ok.

Grammatik:
<meddelande> ::= Kanot <tal><svans> |
<meddelande> kanot <tal><svans>
<svant> ::= och | ska in! | ,
<tal> ::= 1 | 2 | 3 ...

Mening at parsa: Kanot 42 och kanot 4711 ska in! kanot 17,



Felaktig mening som grammatiken godkänner

Grammatik:
<meddelande> ::= Kanot <tal><svans> |
<meddelande> kanot <tal><svans>
<svant> ::= och | ska in!,
<tal> ::= 1 | 2 | 3 ...

5. Värsta webbsyntaxen (Tildatenta 000831)

En webbfil innehåller dels webbsidans text, dels taggar för radbrytningar och indragningar. Taggen
 ger ny rad och för att få indragning av ett textavsnitt skriver man taggen <Q> före och taggen </Q> efter. Exempelvis ger webbfilen Organismer
 <Q> Djur
 <Q> Flugor
 Sillar
 </Q> Svamp
 <Q> Flugsvamp
 Sillkremla
 </Q> </Q> följande webbsideutseende:

Organismer
- Djur
  - Flugor
  - Sillar
- Svamp
  - Flugsvamp
  - Sillkremla

Skriv en syntax för webbfiler där endast dessa taggar och vanlig text före- kommer. Du kan få använda <text> för att beteckna godtycklig taggfri text.

Låt os skriva om det: **websida1**=
Organismer
 <Q>
Djur
 <Q> Flugor
 Sillar
 </Q>
Svamp
 <Q> Flugsvamp
 Sillkremla
 </Q>
</Q>

Men detta är också en websida:
websida2=
Djur
 <Q> Flugor
 Sillar
 </Q>
Svamp
 <Q> Flugsvamp
 Sillkremla
 </Q>

Så vi kan egentligen skriva **websida1** som Organismer **
 <Q> websida2 </Q>**. Så texten mellan <Q> och </Q> är också en websida. Vi får regeln.

(1) **websida=<Q> websida </Q>**

En trivial hemsida är endast en **text**, så vi behöver att grammatiken godkänner det.

(2) **websida=text**

Slutligen observerar vi att om **organismer** är en websida och **<Q> Djur
 <Q> Flugor
 Sillar
 </Q> Svamp
 <Q> Flugsvamp
 Sillkremla
 </Q></Q>** är en websida endlig (1) så får vi att **websida
 websida** är en websida. Det ger oss regeln:

(3) **websida=websida
 websida**

Den resulterande grammatiken blir:

```
<hemsida> ::= <text>
            | <Q><hemsida></Q>
            | <hemsida><BR><hemsida>
<Q>::= "<Q>"
</Q>::= "</Q>"
<BR>::= "<BR>"
```