Preliminärt försättsblad

# Språkgranskning med reguljära uttryck
# Language analysis with regular expressions

Alexander Baltatzis
Uppdragsgivare och handledare: professor Viggo Kann
Examinator: professor Stefan Arnborg

## Sammanfattning

Slutresultatet av detta examensarbete är avsett att kunna utgöra en kompletterande funktion till Granska, ett experimentellt program för datorstödd språkgranskning som utvecklas på KTH. Granska arbetar för närvarande bara med hela ord eller ordsekvenser. För att utöka funktionaliteten till att spåra felaktiga förkortningar, felaktigt formaterade datum m.m. måste teckenföljder kunna analyseras.

För att lösa detta behöver Granska införa stöd för reguljära uttryck. Det finns flera olika verktyg för att implementera stöd för reguljära uttryck i ett språkgranskningsprogram. Dessa verktyg är implementerade som en deterministisk finit automat (DFA) eller en icke-deterministisk finit automat (NFA) eller en kombination av dessa.

En kombinerad DFA- och NFA-lösning samt en renodlad NFA-lösning har undersökts och diskuteras i rapporten. Olika program för att testa och mäta effektivitet har utvecklats. Dessutom har program utvecklats för att generera testmaterial så att spårbarhet och reproducerbarhet kan garanteras.

Resultaten visar att en kombinerad DFA- och NFA-lösning är förkrossande mer effektiv givet Granskas speciella förutsättningar – många men små reguljära uttryck. Den kombinerade DFA/NFA-lösningen är dock mer komplicerad och mer kostsam att underhålla.

## Abstract language analysis with regular expressions

In this Master’s project I have examined different approaches to enhance the functionality of Granska, an advanced computer program for analyzing texts in Swedish grammatically. Currently, Granska works with whole words or sentences and is not able to analyze abbreviations, date formats, numberings, etc. Adding support for regular expressions will enhance Granska to analyze and write grammatical rules for these kinds of problems.

There are several tools available to provide Granska with support for regular expressions. All tools are implemented either as a Deterministic Finite Automata (DFA) or a Non Deterministic Finite Automata (NFA) or a combination of the two. The differences are discussed in the thesis, suffice to say that while a DFA tool is more effective than an NFA, a DFA tool cannot support back references and that is a prerequisite for Granska.

In the thesis, a combined DFA and NFA solution using Flex and GNU Regex is compared with an NFA solution using solely GNU Regex. Given the special conditions of Granska with many but small regular expressions, the combined DFA/NFA solution outshines the NFA solution to the price of a more complex maintenance of the combined solution.

# Innehåll


1 Inledning ..... 4
2 Bakgrund ..... 5
3 Reguljära uttryck ..... 5
3.1 Vad reguljära uttryck används till ..... 7
3.2 Metatecken i reguljära uttryck ..... 7
3.3 Olika syntaxer av reguljära uttryck ..... 7
3.4 Andra metatecken ..... 8
3.5 Grupperingar och alternativ ..... 9
3.6 Bakåtreferenser ..... 9
3.7 Svårtydda reguljära uttryck ..... 10
3.8 Sammanfattning ..... 10
4 Implementering av reguljära uttryck ..... 12
4.1 Automat ..... 13
4.2 Skillnader mellan NFA och DFA ..... 14
4.3 Hungrig matchning ..... 14
4.4 Skillnader i hur alternativ kan matchas ..... 15
4.5 Kompilering av reguljära uttryck ..... 15
4.6 Inga bakåtreferenser i DFA ..... 15
5 Problemställning ..... 15
5.1 Regelfil med exempel ..... 16
5.2 Behövs bakåtreferenser? ..... 16
6 Undersökning ..... 17
6.1 Kombinerad DFA och NFA ..... 17
6.2 Enbart NFA ..... 18
6.3 Frågeställningar ..... 18
6.4 Tillverkning av testfiler ..... 18
6.5 Osäkerheter i mätningarna ..... 19
7 Resultat ..... 19
7.1 Prestandatest DFA/NFA mot NFA ..... 19
7.2 NFA-delen i DFA/NFA-lösningen ..... 20
7.3 Kostnad av vektorallokering ..... 20
7.4 Prestanda som funktion av antal reguljära uttryck ..... 21
7.5 Filstorlek och antal träffar ..... 21
7.6 Kompilatoroptimeringar ..... 24
8 Slutsatser ..... 26
9 Ordförklaringar ..... 27
10 Referenser ..... 28
Bilaga Programdokumentation ..... 29

# 1 Inledning

Slutresultatet av detta examensarbete är avsett att kunna utgöra en kompletterande funktion till Granska, ett experimentellt program för datorstödd språkgranskning som utvecklas på KTH.

Granska som språkgranskningsverktyg arbetar för närvarande bara med hela ord eller ordsekvenser. En närmare presentation av Granska ges i avsnitt 2 *Bakgrund*. För att man skall kunna utöka funktionaliteten till att spåra felaktiga förkortningar, felaktigt formaterade datum m.m. måste ett stöd för att analysera teckenföljder införas.

Det vanligaste sättet att uttrycka regler som känner igen teckenföljder i en text kallas för reguljära uttryck. I avsnitt 3 *Reguljära uttryck* förklaras vad reguljära uttryck är och ges exempel på vad de kan användas till för vanliga datoranvändare.

Det finns flera olika verktyg med vars hjälp man kan implementera stöd för reguljära uttryck i ett språkgranskningsprogram. Dessa verktyg är implementerade som en deterministisk finit automat (DFA) eller en icke-deterministisk definit automat (NFA) eller en kombination av dessa. Begreppen DFA och NFA förklaras närmare i avsnitt 4 *Implementering av reguljära uttryck.*

Syftet med examensarbetet har varit att undersöka om ett DFA- eller NFA-baserat verktyg, alternativt en kombination av de två, skulle lämpa sig bäst för Granska. Att använda sig av DFA är i regel effektivare och snabbare än NFA men DFA är begränsat så till vida att bakåtreferenser inte kan användas när man granskar reguljära uttryck. Eftersom Granska troligtvis har stort behov av bakåtreferenser så är en renodlad DFA-lösning därför inget lämpligt alternativ. Granskas speciella förutsättningar gås igenom i avsnitt 5 *Problemställning.*

De alternativ som har undersökts är dels en kombinerad DFA- och NFA-lösning samt en renodlad NFA-lösning. Olika program för testning och mätning av effektivitet har utvecklats. Dessutom har program utvecklats för generering av testmaterial så att spårbarhet och reproducerbarhet kan garanteras. Detta beskrivs i avsnitt 6 *Undersökning*. Resultaten från undersökningarna visas i avsnitt 7 *Resultat*.

Resultaten visar att en kombinerad DFA- och NFA-lösning är förkrossande mer effektiv givet Granskas speciella förutsättningar. Denna lösning är dock mer komplicerad och antagligen mer kostsam att underhålla. Mer om detta beskrivs i avsnitt 8 *Slutsatser*.

De begrepp som används i detta dokument definieras fortsättningsvis första gången de förekommer i rapporten. I slutet av rapporten återfinns dessutom en lista med ordförklaringar.

Som bilaga bifogas programdokumentation med klassdiagram, sekvensdiagram samt gränssnittsdokumentation.

# 2 Bakgrund

Granska är ett experimentellt program för datorstödd språkgranskning som utvecklas på Nada vid KTH. På Granskas webbsida finns mer utförlig information om detta språkgranskningsprogram. Nedanstående presentation är ett utdrag ur populärbeskrivningen av Granska på dess hemsida www.nada.kth.se/theory/projects/granska/.

> ”Granska ser ut som ett vanligt ordbehandlingsprogram med de bekanta menynamnen Arkiv, Redigera, Format, Verktyg och Hjälp. Liksom i andra ordbehandlingsprogram har användaren tillgång till grundläggande ordbehandlingsfunktioner: skriva, redigera, formatera och spara text. Utöver detta finns en språkgranskningsfunktion, ett hjälpsystem med skrivregler och möjlighet till sökning på ordklasser.
>
> Granskningsfunktionen analyserar språket i texten och markerar de problem den hittar i texten. Exempel på typer av fel som programmet kan hitta är stavfel, felaktigt skrivna tecken (till exempel *Anna's*), stilavvikelser (till exempel *våran*) och grammatikfel (till exempel *en hus*). - - -
>
> Granska använder sig av en uppsättning regler som beskriver vanliga skrivfel. Felaktigheter i texten hittar Granska genom att undersöka om någon regel matchar någon ordsekvens i texten. Ett exempel på en regel är att det inte får förekomma adjektiv i singular följt av substantiv i plural, t.ex. *rund bollar*. För att matchning skall kunna göras måste varje ord i texten förses med ordklassinformation, en så kallad tagg. Först när texten har taggats kan Granska leta efter fel genom att matcha regler mot taggade ordsekvenser.”

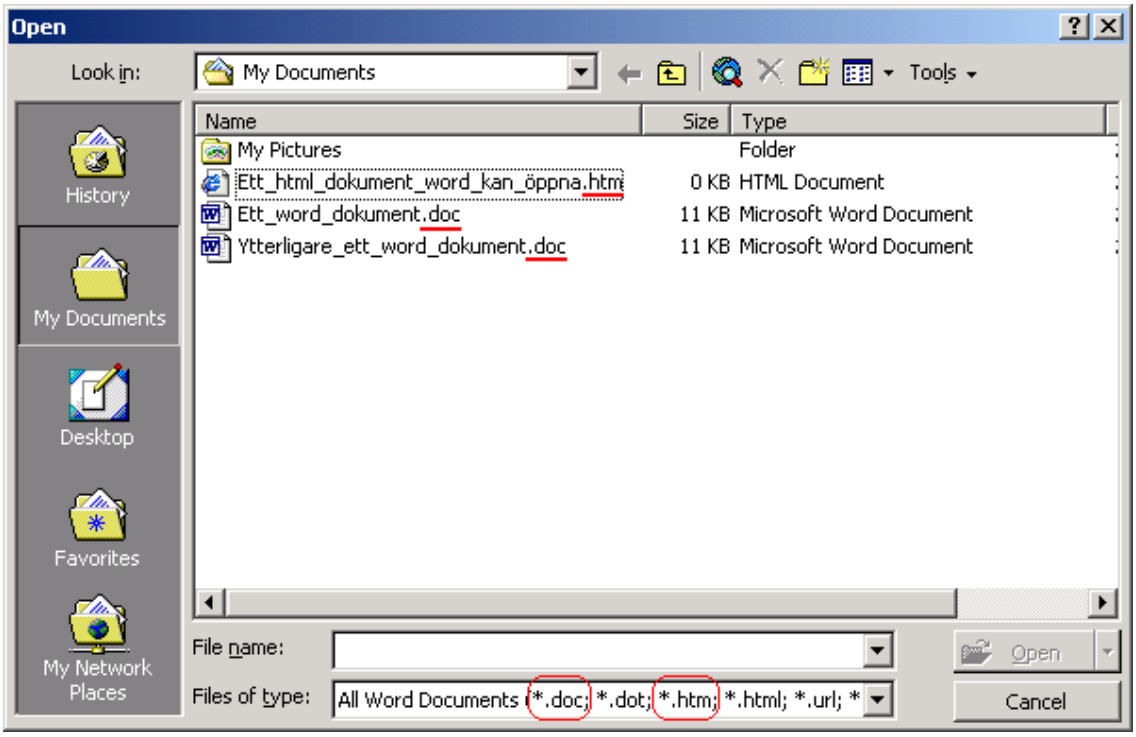
Granska arbetar i nuläget endast med hela ord eller ordsekvenser. För att man skall kunna utöka funktionaliteten till att känna igen felaktiga förkortningar, felaktigt formaterade datum m.m. måste stöd för att analysera teckenföljder införas. Det vanligaste sättet att uttrycka regler för att känna igen teckenföljder i en text kallas för reguljära uttryck.

# 3 Reguljära uttryck

Reguljära uttryck används när man vill söka efter en viss text i en textmassa. Ett reguljärt uttryck består av tecken och metatecken som tillsammans utgör ett sökmönster.

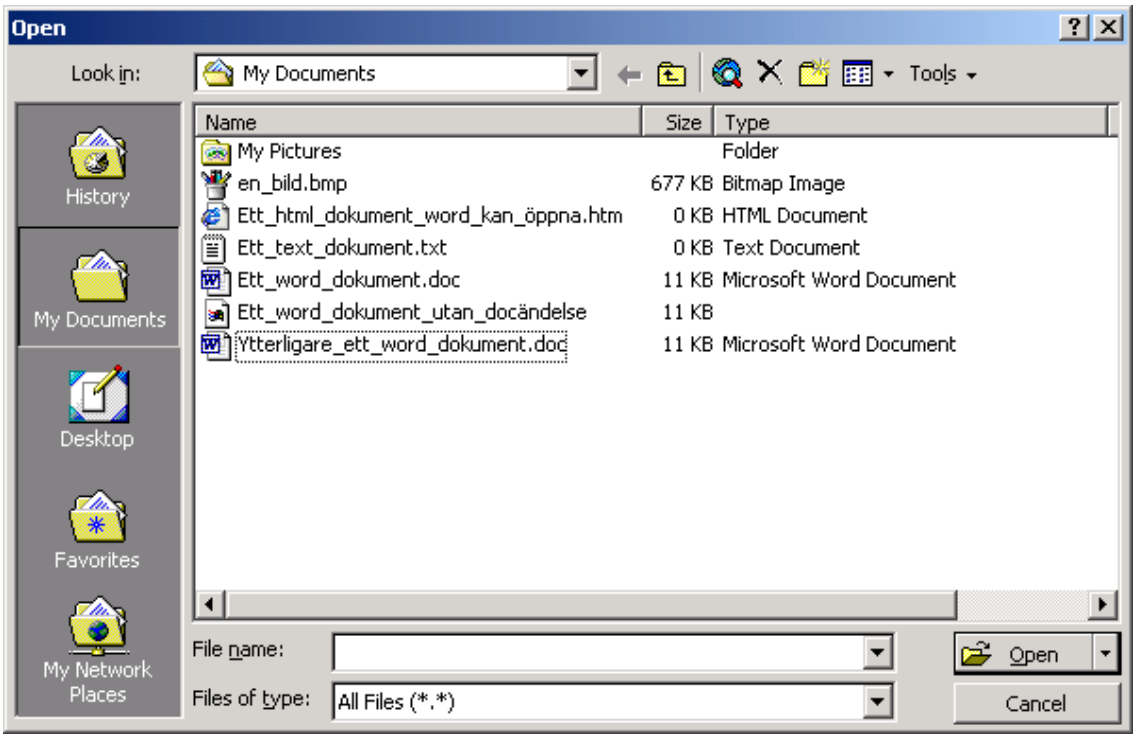
Ett exempel på en vanlig användning av reguljära uttryck är filöppningsdialogen i en ordbehandlare. I *Figur 1* visas en filöppningsdialog i ordbehandlaren Microsoft Word. I fältet *Files of type* räknas ett antal reguljära uttryck upp: ***.doc**, ***.dot**, ***.htm** osv.

I dialogfönstret visas enbart filer vars filnamn **matchar** något av dessa reguljära uttryck. Exempelvis matchar filen *Ett_word_dokument.doc* sökmönstret **.doc*.




**Figur 1 Filöppningsdialog i Word, wordfiler**

Övriga filer i den öppnade mappen visas inte. För att se övriga filer i mappen måste ett annat sökmönster användas. Se *Figur 2*.




**Figur 2 Filöppningsdialog i Word, alla filer**

## 3.1 Vad reguljära uttryck används till

Reguljära uttryck används i *Figur 1* ovan för att man bland alla tillgängliga filer skall kunna välja ut de filer som Word kan öppna.

Det delprogram, hädanefter benämnt *sökmönstermotorn*, som tolkar det reguljära uttrycket är inte medvetet om att det är wordfiler som väljs ut. Sökmönstermotorn betraktar alla filer i mappen som en textmassa:

```
Ett_html_dokument_word_kan_öppna.htm
Ett_text_dokument.txt
Ett_word_dokument.doc
Ett_word_dokument_utan_docändelse
Ytterligare_ett_word_dokument.doc
en_bild.bmp
```

På denna textmassa applicerar sökmönstermotorn de reguljära uttrycken och ett antal rader som matchar sökmönstren skiljs ut:

```
Ett_word_dokument.doc
Ett_html_dokument_word_kan_öppna.htm
Ytterligare_ett_word_dokument.doc
```

Reguljära uttryck används till att specificera kriterier varvid man kan skilja ut teckenföljder ur en textmassa.

## 3.2 Metatecken i reguljära uttryck

Det reguljära uttrycket ***.doc** i *Figur 1* består av ett metatecken ’*’ samt fyra vanliga tecken ’.doc’. Med metatecken avses att tecknet inte matchar sig självt utan skall tolkas särskilt. I det här fallet kommer ’*’ att matcha en teckenföljd. Det reguljära uttrycket kommer således att matcha filnamn bestående av en teckenföljd, vilken som helst, avslutat med ändelsen ’.doc’. Exempel:

```
Ett_word_dokument.doc
Ett_till_dokument.doc
Ett_.doc_dokument.doc
```

## 3.3 Olika syntaxer av reguljära uttryck

I andra sammanhang, t.ex. när man använder programmeringsverktyg som lex, flex eller perl, har metatecknet ’*’ en liknande men annan tolkning. Lex sökmönstermotor tolkar ’*’ som *inga eller flera av föregående* och kan appliceras på ett enskilt tecken eller ett helt reguljärt uttryck.

För att belysa skillnaden i syntax, antag att det reguljära uttrycket **Ett*.doc** används för att välja ut filer i Words filväljare respektive sökmönstermotorn i lex.

Filväljaren i Word kommer att matcha filnamn som börjar på ’Ett’ och slutar på ’.doc’. Exempel:

```
Ett_word_dokument.doc
```

Lex sökmönstermotor kommer däremot inte att matcha ovanstående exempel eftersom det reguljära uttrycket tolkas annorlunda nämligen att matcha filnamn som börjar på ’E’ följt av ett eller flera ’t’ följt av ’.doc’. Exempel:

Ettttt.doc

Det är viktigt att konstatera att **tolkningen av reguljära uttryck kan vara olika i olika applikationer och verktyg.** Samma sökmönster kan matcha olika teckenföljder i olika sökmönstermotorer.

En jämförelse mellan olika sökmönstermotorer med avseende på t.ex. prestanda är missvisande om inte tolkningen av ett reguljärt uttryck är densamma.

I Granska skall reguljära uttryck matcha felskrivningar av förkortningar m.m. Det är viktigt att testa att de reguljära uttrycken fungerar som avsett, dvs. att felaktigheter hittas och att inte korrekt text matchas av misstag.

## 3.4 Andra metatecken

Det finns ofta flera metatecken till sökmönstermotorer och metatecknens betydelse kan, som nämnts, variera. För förståelse av den fortsatta rapporten måste de metatecken som används i rapporten entydigt definieras. Syntaxen följer POSIX reguljära uttryck såsom de skrivs i t.ex. UNIX-programmen lex och grep.

Metatecknet ’*’ har redan nämnts. Det kommer fortsättningsvis ha betydelsen: *inga eller flera av föregående* vilket exemplifierades i föregående avsnitt. Metatecknet ’+’ har en liknande definition nämligen *minst ett eller flera av föregående*. Metatecken som opererar på föregående brukar kallas kvantifierare.

Metatecknet ’.’ används för att matcha ett tecken vilket som helst. Exempel:

| Reguljärt uttryck | Exempel på matchande text |
|---|---|
| `b..a` | `bada` |
| `b..a` | `boka` |
| `s.*r` | `sommar` |

Man definierar en teckenmängd genom att omsluta mängden med ’[’ och ’]’. Inom dessa klamrar kan man använda bindestreck ’-’ för att lättare definiera en teckenföljd.

| Reguljärt uttryck | Exempel på matchande text |
|---|---|
| `[ab][lo]` | `al` |
| `[ab][lo]` | `bo` |
| `[0-9]` | `1` |

Mängder går att kombinera med ’*’ eller ’+’.

| Reguljärt uttryck | Exempel på matchande text |
|---|---|
| `[a-z]*` | `dokument` |
| `[1-2]*` | `1122` |
| `[0-9]+` | `1234` |

## 3.5 Grupperingar och alternativ

Metatecknet ’|’ betecknar alternativ.

| Reguljärt uttryck | Exempel på matchande text |
|---|---|
| `a\|b` | `a` |
| `a\|b` | `b` |

Som kommer att framgå har alternativ stor betydelse för hur reguljära uttryck implementeras.

Metatecknen ’(’ och ’)’ definierar en gruppering i ett reguljärt uttryck. En gruppering kan användas när man vill definiera prioritet (jämför med matematiska uttryck som 2*(1+2)). Exempel på reguljära uttryck med grupperingar:

| Reguljärt uttryck | Exempel på matchande text |
|---|---|
| `de(n\|t)` | `den` |
| `de(nn\|tt\|ss)a` | `dessa` |
| `(s.*\|v.*)r` | sommar |

## 3.6 Bakåtreferenser

En annan mycket viktig funktionalitet som kan erhållas med gruppering är bakåtreferens. Bakåtreferenser kan man använda till att med hjälp av ett reguljärt uttryck söka och ersätta text. En bakåtreferens betecknas fortsättningsvis ’*n*’ där *n* är ordningsnumret på grupperingen. Exempel:

Följande reguljära uttryck skulle kunna användas av Granska för att hitta ett glömt mellanslag mellan en siffra och en specificerad förkortning.

```
([0-9])(kr|km|m|dm|cm|mm|l|dl|cl|ml|kg|hg|tsk|tim|min|sek|år)
```

Text som matchar sökmönstret skall ersättas med hjälp av följande reguljära uttryck.

```
\1 \2
```

Dvs. text som matchar första grupperingen följt av ett mellanslag följt av text som matchar andra grupperingen.

Applicerat på följande text

```
Vispa 2 ägg och 2dl socker. Blanda 3dl mjöl, 1,5tsk bakpulver
och 2 tsk vaniljsocker. Rör ner blandningen...
```

Resulterar matchningen i den nya texten

```
Vispa 2 ägg och 2 dl socker. Blanda 3 dl mjöl, 1,5 tsk
bakpulver och 2 tsk vaniljsocker. Rör ner blandningen...
```

Att söka och ersätta text med hjälp av reguljära uttryck är vanligt i avancerade ordbehandlare som t.ex. GNU emacs. Granska skulle också kunna använda sig av bakåtreferenser för att markera felaktigheten eller för att ge förslag till rättning.

## 3.7 Svårtydda reguljära uttryck

Ett reguljärt uttryck kan lätt formuleras så krångligt att den ursprungliga tanken med det reguljära uttrycket inte är uppenbar. Som tidigare nämnts kan tolkningen av ett reguljärt uttryck vara olika i olika sökmönstermotorer. Om ett reguljärt uttryck skall skrivas om för att fungera i en annan sökmönstermotor är det viktigt med dokumentation om vad sökmönstret var tänkt att matcha samt ett eller flera exempel.

I den mycket underhållande boken *Mastering Regular Expressions* av J.Friedl [3] finns ett exempel på ett monstruöst reguljärt uttryck som matchar alla möjliga varianter av e-postadresser. Exemplet är återgivet i *Figur 3* på nästa sida. Det reguljära uttrycket är skrivet för perl och ser vid första anblicken inte så enkelt ut att anpassa för en annan sökmönstermotor.

## 3.8 Sammanfattning

Ett reguljärt uttryck består av tecken och metatecken som tillsammans utgör ett sökmönster. Sådana sökmönster används dagligen i vanligt förekommande applikationer för att söka efter matchande text. Tolkningen av ett sökmönster kan variera mellan applikationer.

En formell beskrivning av en vanlig notation av reguljära uttryck finns att läsa i manualen till Regex på UNIX-system [11]. Manualen finns fritt tillgänglig på ett flertal platser på internet och kan nås genom sökning på sökorden *man(7) regex*.

Manualen beskriver en POSIX standard för notation av reguljära uttryck *POSIX 1003.2, section 2.8 (Regular Expression Notation).*

```
[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-
\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:(?:[^(\040)<>@,;:".\\
\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?
:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*")[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\x
ff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()
]*)*\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff
\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?
:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"[^\\\
x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*")[\040\t]*(?:\([^\\\x80-\xff\n\015()
]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^
\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*@[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff
]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*
)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\
x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()
]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^
\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\x
ff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()
]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037
\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015(
)]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[
^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*|(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\04
0)<>@,;:".\\\[\]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\01
5"]*)*")[^()<>@,;:".\\\[\]\x80-\xff\000-\010\012-\037]*(?:(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[
^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff
\n\015()]*)*\)|"[^\\\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*")[^()<>@,;:".\\
\[\]\x80-\xff\000-\010\012-\037]*)*<[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\
([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\
)[\040\t]*)*(?:@[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015
()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\0
40)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\[(?:[^\\\x8
0-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|
\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*
\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\0
15()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(
\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\[(?:[^\\\
x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff
]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*
)*\)[\040\t]*)*(?:,[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\
n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*@[\
040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\x
ff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\]\
000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|
\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\
015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:\.
[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-
\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\
]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]
]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\
n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)
*:[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x8
0-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)?(?:[^(\040)<>@,;:".\
\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"[^\\\x80-\xff\n\015"]*(
?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*")[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\
xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015(
)]*)*\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xf
f\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(
?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"[^\\
\x80-\xff\n\015"]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015"]*)*")[\040\t]*(?:\([^\\\x80-\xff\n\015(
)]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[
^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*@[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xf
f]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]
*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037
\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015(
)]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[
^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\
xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015(
)]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\0
37\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\01
5()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\)
)[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*>)
```

**Figur 3 Reguljärt uttryck för matchning av en e-postadress**

# 4 Implementering av reguljära uttryck

I förra kapitlet beskrevs hur reguljära uttryck tolkas av olika sökmönstermotorer. I detta kapitel beskrivs hur en sökmönstermotor bär sig åt för att matcha text mot ett sökmönster.

Det finns i princip två metoder för matchning av reguljära uttryck. Nedanstående exempel illustrerar skillnaderna i tillvägagångssätt mellan dessa två metoder.

Antag att texten *köksstolen* skall matchas mot det reguljära uttrycket **köks(stol|bord|bänk|stolen)**.

| Text | Reguljärt uttryck |
|---|---|
| `köksstolen` | `köks(stol\|bord\|bänk\|stolen)` |

*Metod 1*

Denna metod kommer att först matcha de fyra första bokstäverna *köks* i tur och ordning. Därefter kommer alternativen att prövas i tur och ordning. *Köksstol* matchar nästan men inte riktigt. Sökmönstermotorn måste nu backa tillbaka till de fyra första bokstäverna *köks*. Nästa alternativ *köksbord* matchar inte alls, inte heller *köksbänk*. Det sista alternativet matchar däremot. Förloppet illustreras nedan:

| Text | Reguljärt uttryck |
|---|---|
| **köks**stolen | **köks**(stol\|bord\|bänk\|stolen) |
| **köksstol**en | **köks**(**stol**\|bord\|bänk\|stolen) |
| **köks**stolen | **köks**(~~stol~~\|bord\|bänk\|stolen) |
| **köks**stolen | **köks**(~~stol~~\|~~bord~~\|bänk\|stolen) |
| **köksstolen** | **köks**(~~stol~~\|~~bord~~\|~~bänk~~\|**stolen**) |

*Metod 2*

Metod 2 börjar med att, på samma sätt som metod 1, matcha de fyra första bokstäverna *köks*. Därefter matchas ytterligare en bokstav – *kökss* – då faller de båda alternativen *köksbord* och *köksbänk* bort. Efter ytterligare fyra bokstäver – *köksstole* – faller alternativet *köksstol* bort och slutligen matchas det sista alternativet *köksstolen*. Förloppet illustreras nedan:

| Text | Reguljärt uttryck |
|---|---|
| **köks**stolen | **köks**(stol\|bord\|bänk\|stolen) |
| **köks**stolen | **köks**(stol\|~~bord~~\|~~bänk~~\|stolen) |
| **köksstole**n | **köks**(~~stol~~\|~~bord~~\|~~bänk~~\|**stolen**) |
| **köksstolen** | **köks**(~~stol~~\|~~bord~~\|~~bänk~~\|**stolen**) |

Man kan misstänka att metod 2 är snabbare än metod 1 eftersom den inte behöver backa tillbaka i texten. Den kräver dock en mera avancerad förbehandling av det reguljära uttrycket.

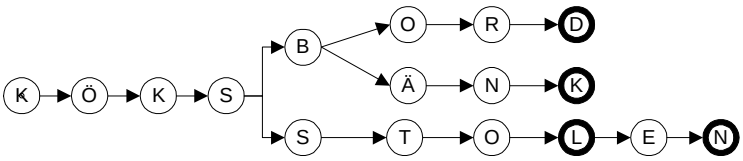
Metod 1 är implementerad med en icke-deterministisk finit automat (NFA[1]) och metod 2 med en deterministisk finit automat (DFA[2]). Vad är då en automat?

[1] Non Deterministic Finite Automata

## 4.1 Automat

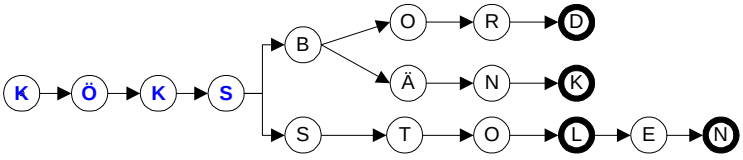
En analysator för ett reguljärt uttryck kan beskrivas med ett tillståndsdiagram. Ett sådant diagram beskriver olika tillstånd som sökmönstermotorn kan anta när den analyserar en text.

Ett enkelt tillståndsdiagram för metod 2 och sökmönstret **köks(stol|bord|bänk|stolen)** visas nedan. För varje bokstav i ordet som skall undersökas, ändras tillståndet (ett steg åt höger). Om bokstaven inte stämmer matchar inte ordet. När det undersökta ordet är slut skall tillståndet ha stannat vid någon av de fetmarkerade ringarna. De fetmarkerade ringarna är accepterande tillstånd, d.v.s. då ordet matchar sökmönstret. De ord som matchar detta sökmönster är köksbord, köksbänk, köksstol och köksstolen.

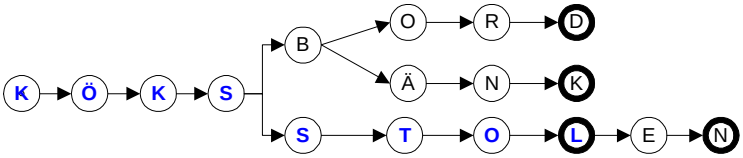



Förloppet för metod 2 med exemplet *köksstolen* blir då enligt:

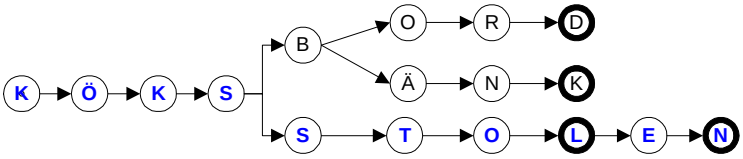
**köks**stolen




**köksstol**en




**köksstolen**




En automat är ett annat ord för ett strikt definierat tillståndsdiagram.

[2] Deterministic Finite Automata

## 4.2 Skillnader mellan NFA och DFA

Skillnaden mellan NFA och DFA ligger i hur de behandlar alternativ i reguljära uttryck. Ett reguljärt uttryck består av ett eller flera alternativ (åtskilda av ’|’ , se avsnitt 3.5).

Det går alltid att bilda en NFA av ett reguljärt uttryck genom att dela upp det i dess beståndsdelar och därefter bena ut varje beståndsdel eller förgrening. Detta förfarande finns beskrivet i kapitel två i boken *Compiler Construction* [9].

I samma bok (i appendix) finns också ett bevis för att det går att bilda en DFA från en NFA. En DFA bildas i princip genom hopslagning av alla alternativa grenar i NFA, så att det för varje tecken i texten som undersöks endast finns en möjlig gren. Därav namnet *deterministisk* automat, för varje tecken som undersöks finns bara ett ”ödesbestämt” alternativ.

I en NFA går det inte alltid att, givet ett tecken, avgöra säkert vilket alternativ som matchar förrän ytterligare tecken undersöks. Därav namnet icke-deterministisk automat.

En annan skillnad är effektivitet. En DFA:s operationer växer linjärt med storleken på texten som skall genomsökas eftersom dessa tecken läses in ett i taget utan att automaten backar. Det går inte att säga detsamma om NFA. Det finns exempel på reguljära uttryck som har exponentiell komplexitet för NFA-baserade sökmönstermotorer. Detta förklaras av A.V. Aho [1].

## 4.3 Hungrig matchning

Kvantifierare (se avnitt 3.4) som ’+’ och ’*’ matchar normalt så mycket som möjligt av föregående tecken eller reguljära uttryck. Detta kallas *hungrig* matchning.

Antag att det reguljära uttrycket **Vispa (.*) dl** söks efter i texten:

```
Vispa 2 ägg och 2 dl socker. Blanda 3 dl mjöl, 1,5 tsk
bakpulver och 2 tsk vaniljsocker. Rör ner blandningen...
```

I de flesta sökmönstermotorer matchas:

```
Vispa 2 ägg och 2 dl socker. Blanda 3 dl mjöl, 1,5 tsk
bakpulver och 2 tsk vaniljsocker. Rör ner blandningen...
```

Flera NFA-baserade sökmönstermotorer som t.ex. perl har ytterligare metatecken så att man kan styra detta *hungriga* beteende och i stället matcha:

```
Vispa 2 ägg och 2 dl socker. Blanda 3 dl mjöl, 1,5 tsk
bakpulver och 2 tsk vaniljsocker. Rör ner blandningen...
```

I de reguljära uttryck som Granska använder har denna typ av detaljstyrning av reguljära uttryck inte behövts.

### 4.4 Skillnader i hur alternativ kan matchas

Alternativ kan matchas olika, antingen matchas längsta möjliga alternativ eller först funna alternativ. I början av kapitlet visades ett exempel med det reguljära uttrycket **köks(stol|bord|bänk|stolen)** som söks efter i texten *köksstolen*. Det visades hur hela ordet *köksstolen* till slut matchade. För sökmönstermotorn gällde implicit att längsta möjliga matchning vinner. Så fungerar t.ex. lex (DFA) och POSIX Regex (NFA).

En del sökmönstermotorer som t.ex. perl väljer istället av prestandaskäl att första möjliga match vinner. Samma exempel som tidigare:

| Text | Reguljärt uttryck |
|---|---|
| **köks**stolen | **köks**(stol\|bord\|bänk\|stolen) |
| **köksstol**en | **köks**(**stol**\|bord\|bänk\|stolen) |

Granska bör inte matcha på detta sätt, ty om Granska skall markera en språklig felaktighet måste den fullständiga felaktigheten markeras.

### 4.5 Kompilering av reguljära uttryck

Innan en sökmönstermotor kan känna igen ett reguljärt uttryck måste automaterna bildas. Automaterna representeras av en datastruktur i minnet.

Det finns DFA-baserade sökmönstermotorer som exempelvis lex och flex som kan kompilera denna minnesstruktur i förväg med programspråken C eller C++. Detta passar Granska bra eftersom det är skrivet i C++.

Kompilering i förväg gör att sökmönstermotorn matchar snabbare eftersom en del av det jobb sökmönstermotorn skall göra, nämligen att bilda en automat, är gjort i förväg. Granska har ungefär 90 reguljära uttryck. Lex/flex kommer att bilda en stor automat för att känna igen samtliga regler.

### 4.6 Inga bakåtreferenser i DFA

En nackdel med DFA är att bakåtreferenser (se avsnitt 3.6) inte kan användas [3]. Vissa DFA-baserade sökmönstermotorer som t.ex. grep använder sig av NFA för att tillhandahålla bakåtreferenser då det efterfrågas.

## 5 Problemställning

Vid testning är det viktigt att ta hänsyn till Granskas förutsättningar och tänkta användningsområde. Textfilerna som skall genomsökas kan antagligen vara väldigt stora. Inga groteskt stora sökmönster som i avsnitt 3.7 kommer dock att behövas. De sökmönster som behövs för att matcha språkliga felaktigheter är relativt små men många till antalet.

## 5.1 Regelfil med exempel

Till detta examensarbete har en experimentell uppsättning reguljära uttryck som matchar felaktiga förkortningar, felaktigt skrivna datum m.m. arbetats fram. Att formulera dessa regler kräver stor kunskap om svenska språket och de är utarbetade av språkforskare vid Kungliga Tekniska Högskolan.

Reglerna är samlade i en regelfil tillsammans med exempel på den felaktighet regeln är avsedd att matcha samt ett exempel på det rätta skrivsättet som inte skall matchas. I nuläget är det 91 regler. Nedan ett regelexempel avsett att fånga upp felaktiga skrivningar av förkortningen *p.g.a*.

```
[ ]p((ga\.)|(ga)|(g a)|(ga)|(\.ga\.)|(\.ga)|(g\.a\.)|(g\.a))
```

Som nämnts i kapitel 3 kan reguljära uttryck tolkas olika av olika sökmönstermotorer. Exemplen i regelfilen har använts för att kontrollera att olika sökmönstermotorer matchar rätt. Ibland har ett sökmönster behövt skrivas om för att kunna användas. Det har i vissa fall varit väldigt svårt att förstå hur regeln skall skrivas om (se avsnitt 3.7 *Svårtydda reguljära uttryck*).

## 5.2 Behövs bakåtreferenser?

Det reguljära uttrycket **[0-9]:?(dje|nde|de|te)[^a-zåäöA-ZÅÄÖ]** är tänkt att matcha felaktigheter som *3:dje* och *7nde* men inte det korrekta *3:e* och *7:e*. Om Granska vill markera felaktigheten, t.ex. med färgläggning och understrykning, så görs detta enkelt med bakåtreferens (se avsnitt 3.6), i det här fallet ’**\1**’. Exempel:

```
3:dje
7nde
```

Det skulle också gå att räkna ut ungefär vad felaktigheten i den matchande texten bestod i utan bakåtreferenser genom att betrakta sökmönstret. Det första tecknet som matchar uttrycket är en siffra och det sista tecknet som matchar är inte en bokstav (troligen ett blankslag). Mellanliggande tecken utgör felaktigheten, möjligen med undantag av ett korrekt kolon, se nedan:

```
3:dje
7nde
```

Med bakåtreferens kan man alltid markera felaktigheter exakt genom att använda parenteser i det reguljära uttrycket.

Huruvida det skulle gå att klara sig utan bakåtreferenser beror på hur komplexa regler Granska behöver. Det ligger utanför denna rapport att definiera de regler som Granska behöver för att komplettera dess rättstavningsregler.

En stor poäng med bakåtreferenser är att det enkelt går att definiera ett metaspråk för markering av felaktigheter samt förslag till rättning.

# 6 Undersökning

Eftersom bakåtreferenser är ett krav i uppdragsbeskrivningen för detta arbete så behövs NFA. De två lösningar som jämförs är dels en kombinerad DFA- och NFA-lösning (DFA/NFA), dels en renodlad NFA-lösning. I valet av verktyg beaktas också kravet på plattformsoberoende, åtminstone Windows och Unix skall stödjas.

## 6.1 Kombinerad DFA och NFA

I denna lösning används programmeringsverktyget flex (DFA) för att söka igenom en textfil. Varje träff läggs in i en lista (C++ standardvektor). För att få bakåtreferenser görs ånyo en sökning på den redan matchade träffen med GNU Regex (NFA). Båda dessa verktyg är gratis och finns på ett antal olika plattformar.

Denna lösning består av flera olika delar och delprogram som måste byggas i rätt ordning.

För att bygga testprogrammet måste DFA först vara byggd med flex. För att bygga DFA med flex måste en lexfil ha genererats. Lexfilen genereras med ett egenutvecklat delprogram (generatelex.exe) genom extrahering av reglerna från regelfilen. Byggordningen blir följande:

1. generatelex.exe byggs,
2. generatelex.exe körs och genererar med hjälp av regelfilen en lexfil,
3. flex körs med lexfilen och skapar DFA i en objektfil och
4. huvudprogrammet byggs och länkas med ovanstående objektfil.

Programmet make används för att hålla reda på byggordningen. Regelfilen används två gånger, dels för att bygga en DFA med flex som länkas in i programmet, dels för att läsas in under exekvering till en regelvektor som används av GNU Regex. Därmed finns det risk för synkroniseringsproblem om regelfilen ändras utan att omkompilering sker. Ett sätt att lösa detta är att alltid köra makefilen innan huvudprogrammet körs med t.ex. ett tvåradersskript.

Som nämnts i avsnitt 3 kan tolkningen av reguljära uttryck skilja sig. GNU Regex och flex tillämpar dock liknande tolkningar vilket är en fördel. T.ex. så matchas alltid längsta möjliga alternativ (se avsnitt 4).

Vissa syntaxskillnader finns. För att lösa dessa syntaxskillnader så anpassas reglerna till flex i generatelex.exe (`RegRuleParser::modifyRegexToLex`) som genererar lexfilen. Det är alltså inte exakt samma uppsättning regler som används av GNU Regex och flex. Felsökning av t.ex. felaktigt skrivna reguljära uttryck blir mer komplicerad.

Ett testprogram mäter och redovisar tiden det tar för DFA att hitta alla träffar. Därefter gås alla träffar igenom och första bakåtreferens efterfrågas vilket aktiverar NFA-delen. Den sammanlagda tiden redovisas.

## 6.2 Enbart NFA

Till denna lösning används enbart biblioteket GNU.

Konstruktionen är okomplicerad, reglerna extraheras från regelvektorn och läses in i en lista (C++ standardvektor).

Filhantering av den textfil som skall undersökas måste skrivas. I den förra lösningen genererar flex kod som tar hand om filhanteringen.

Varje regel appliceras i tur och ordning och om en matchning hittas börjar sökningen om från den matchande textens slut. När sökningen är klar redovisas tidsåtgången.

Det finns snabbare NFA-verktyg som matchar första funna alternativ istället för längsta möjliga alternativ vilket beskrivs i avsnitt 4.4. Men som också nämns i samma avsnitt bör Granska matcha hela felaktigheten och inte enbart delar av den.

## 6.3 Frågeställningar

Prestanda är viktigt för Granska. Att använda sig av DFA är för det mesta snabbare än NFA. Hur mycket snabbare är DFA/NFA-lösningen jämfört med NFA-lösningen? Är skillnaden signifikant? Hur lång tid tar respektive del i DFA/NFA?

Hur mycket påverkar testfilernas storlek snabbheten hos de båda lösningarna? Hur mycket mer tid tar det att söka igenom en dubbelt så stor testfil? Hur påverkas snabbheten av en ändring i antalet reguljära uttryck?

Om filstorleken är konstant men antal träffar fördubblas, hur mycket längre tid tar det? Ibland kan det innebära lika mycket jobb att söka igenom en teckenföljd som bara nästan matchade. Om de exempel som inte skall matcha de korrekta exemplen i regelfilen läggs till i testfilen, hur mycket mer tid tar det då?

## 6.4 Tillverkning av testfiler

För att man skall kunna utföra testerna behövs textfiler som indata. För att man skall kunna svara på frågor på om hur antalet träffar eller antal korrekta regelexempel påverkar prestanda har inom ramen för detta arbete ett program skrivits som genererar testfiler. Testfilsgenereringsprogrammet tar som indata en fil där följande storheter upprepas för varje testfil som skall genereras:

- önskad filstorlek
- antal exempel på felaktigheter som skall matchas
- antal exempel på korrekt text som inte skall matchas.

Programmet extraherar exempel som skall matcha respektive inte matcha från regelfilen. För att få den önskvärda storleken på testfilen fylls filen även på med icke-matchande ord.

För att inte enbart förlita sig på konstruerade testfiler har programmen testats också på en mycket stor (1476 kB) artikelsamling från olika dagstidningar. En sådan stor textsamling kallas korpus och används vid testning av grammatikregler. Problematiken är densamma när man testar grammatikregler, man vill skapa regler för att fånga felaktiga skrivsätt men släppa igenom korrekta skrivsätt. Detta beskrivs närmare i en uppsats av Domeij, Knutsson, Carlberger, och Kann [2]. Vikten av korpus och dess sammansättning beskrivs av Viktoria Svensson [4].

### 6.5 Osäkerheter i mätningarna

Testprogrammen läser text från fil in till en buffert och matchar sedan texten i bufferten mot de reguljära uttrycken. Den del som läser in text från fil till en buffert skiljer sig åt. I flex är den delen inbyggd och endast en filpekare behöver skickas. Regexprogrammet har däremot separat kod för detta ändamål. Implementeringarna kan vara olika och påverka effektiviteten åt ena eller andra hållet.

De reguljära uttrycken är inte utförligt testade för inkonsekvens. Under de tester som gjorts har t.ex. förekommit fall där sökmönstren matchat text de inte varit avsedda att matcha. Ofta har det berott på regler som beskriver felaktiga förkortningar. Det en regel ansett vara felaktig kan vara rätt i en annan regel som beskriver en annan förkortning med en annan betydelse.

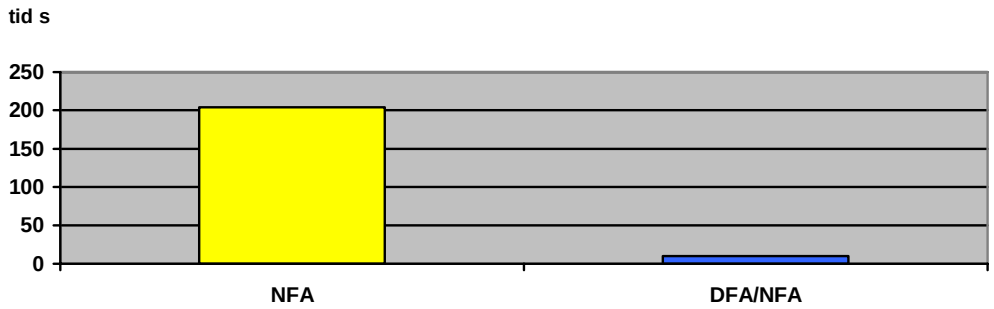
Det som mäts är klockcykler men för att få en bättre bild räknas tidsvärdet ut och redovisas. Den dator som använts för testning har en klockfrekvens på 650 MHz. Givetvis fås andra tider med en annan typ av dator.

## 7 Resultat

### 7.1 Prestandatest DFA/NFA mot NFA

Första testet går ut på att testa de båda metoderna på korpus, den stora artikelsamlingen (1476 kB). Det som mäts är tiden det tar att hitta alla språkliga felaktigheter, alltså enbart DFA-delen i DFA/NFA-lösningen.

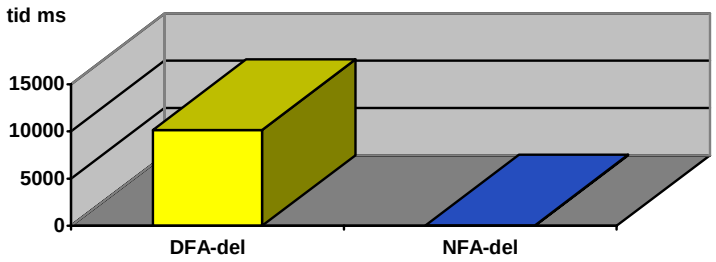
Med DFA/NFA-lösningen tog det 10174 ms (10 sekunder) att hitta 504 träffar. Den rena NFA-lösningen behövde 204303 ms (3 minuter och 24 sekunder) för att hitta samma träffar. Resultaten illustreras i *Figur 4*.




**Figur 4 Regex jämfört med Lex testat på 1476 kB korpus**

## 7.2 NFA-delen i DFA/NFA-lösningen

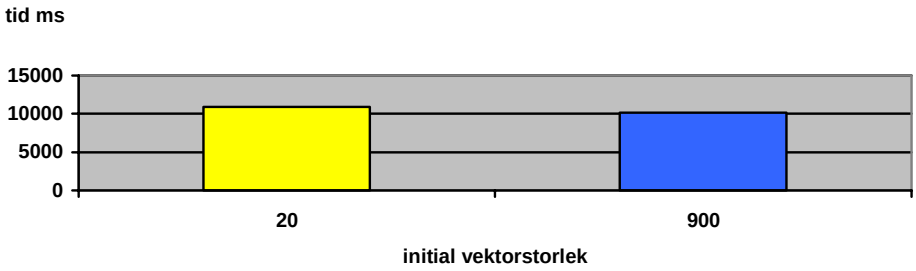
Resultaten visar en förkrossande prestandaskillnad till DFA/NFA-lösningens fördel. Men NFA-delen som hanterar bakåtreferenser är inte aktiverad än. För att mäta detta jobb gås alla träffar igenom och samtliga bakåtreferenser i de regler som hittats efterfrågas. Detta visar sig ta 60 ms. Skillnaden mellan att hitta alla träffar (10 sekunder) med DFA och att gå igenom dem med NFA (60 ms) illustreras i *Figur 5*.




**Figur 5 DFA/NFA-lösningen: DFA-delen jämfört med NFA-delen testat på 1476 kB korpus. NFA-delen är så liten i förhållande till DFA-delen att resultaten måste visas i 3D.**

## 7.3 Kostnad av vektorallokering

DFA/NFA-lösningen sparar undan alla träffar i en C++ standardvektor. En sådan vektor fungerar så att ett sammanhängande minnesutrymme allokeras för att rymma vektorn. Att addera till slutet på vektorn är väldigt billigt. Om utrymmet inte räcker till måste ett större minnesutrymme allokeras och hela vektorn kopieras dit. Denna omallokering kan kosta en del. För att mäta detta prövas ett par olika initiala vektorstorlekar. Med en initial vektorstorlek på 20 tar det 10905 ms att hitta de 504 träffarna. Med vektorstorlek 900 (ingen omallokering) tar det 10164 ms. Skillnaden illustreras i *Figur 6*:




**Figur 6 DFA/NFA-lösningen: Vektorallokeringens kostnad**

## 7.4 Prestanda som funktion av antal reguljära uttryck

Prestandaskillnaden mellan den rena NFA-lösningen och den kombinerade DFA/NFA-lösningen är som visats väldigt stor. Hur mycket påverkar då antalet reguljära uttryck prestanda? Några olika tester har gjorts.

**NFA**

| | | |
|---|---|---|
| Alla 91 regler | 204303 ms | (504 träffar) |
| De första 45 reglerna | 90079 ms | (503 träffar) |
| De 45 sista reglerna | 118039 ms | (1 träff) |
| De 10 första reglerna | 37093 ms | (484 träffar) |
| De 3 första reglerna | 5818 ms | (1 träff) |

**DFA/NFA**

| | | |
|---|---|---|
| Alla 91 regler | 10174 ms | (504 träffar) |
| De första 45 reglerna | 10164 ms | (503 träffar) |
| De 45 sista reglerna | 210 ms | (1 träff) |
| De 10 första reglerna | 250 ms | (484 träffar) |
| De 3 första reglerna | 250 ms | (1 träff) |

Det står klart att mängden reguljära uttryck påverkar prestanda mycket. Eftersom olika regler kan innebära mer eller mindre jobb går det inte att se ett klart samband. Notera t.ex. tidsskillnaden mellan att använda de första respektive sista 45 reglerna.

## 7.5 Filstorlek och antal träffar

För att undersöka hur andra parametrar, såsom filstorlek på indata och antal träffar, påverkar prestanda har tre filstorlekar genererats på 50kB, 300kB, 600kB. För varje filstorlek har genererats dels tre filer preparerade med 100, 300 samt 600 träffar, dels ytterligare tre filer som därutöver också har preparerats med 100, 300 samt 600 exempel på korrekt skrivsätt som inte skall ge träff. De olika kombinationerna av matchande exempel (träffar) och icke-matchande exempel benämns **A-F** i tabellerna nedan. Återigen mäts enbart DFA-delen i DFA/NFA-lösningen, alltså att hitta träffarna. Resultaten illustreras grafiskt i *Figur 7 och 8*.

**Mätningar med renodlad NFA-lösning**

**50 kB filstorlek**

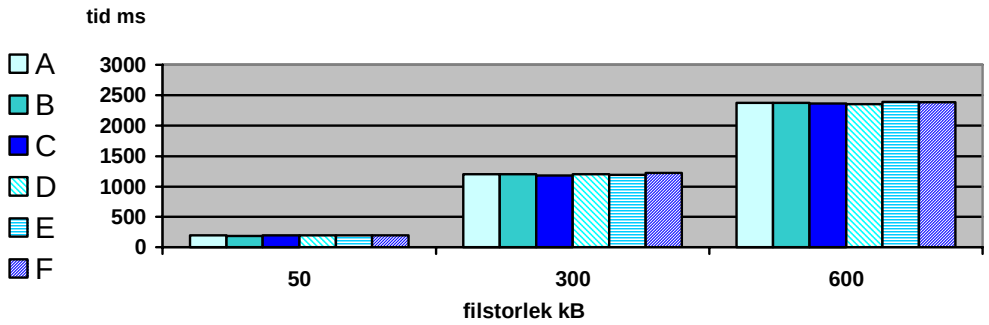
| | Antal preparerade träffar | Antal korrekta exempel som inte skall matcha | Tid |
|---|---|---|---|
| **A** | 100 | | **200** |
| **B** | 300 | | **190** |
| **C** | 600 | | **200** |
| **D** | 100 | 100 | **200** |
| **E** | 300 | 300 | **200** |
| **F** | 600 | 600 | **200** |

**300 kB filstorlek**

| | Antal preparerade träffar | Antal korrekta exempel som inte skall matcha | Tid |
|---|---|---|---|
| **A** | 100 | | **1201** |
| **B** | 300 | | **1201** |
| **C** | 600 | | **1181** |
| **D** | 100 | 100 | **1201** |
| **E** | 300 | 300 | **1191** |
| **F** | 600 | 600 | **1221** |

**600 kB filstorlek**

| | Antal preparerade träffar | Antal korrekta exempel som inte skall matcha | Tid |
|---|---|---|---|
| **A** | 100 | | **2373** |
| **B** | 300 | | **2373** |
| **C** | 600 | | **2383** |
| **D** | 100 | 100 | **2353** |
| **E** | 300 | 300 | **2393** |
| **F** | 600 | 600 | **2383** |




**Figur 7 NFA-lösningens resultat testat på genererade testfiler**

Det finns ett tydligt samband mellan filstorlek och tid. Man kan ana sig till ett linjärt samband. Antal träffar ser inte ut att göra någon större skillnad.

**Mätningar med DFA/NFA-lösning**

**50 kB filstorlek**

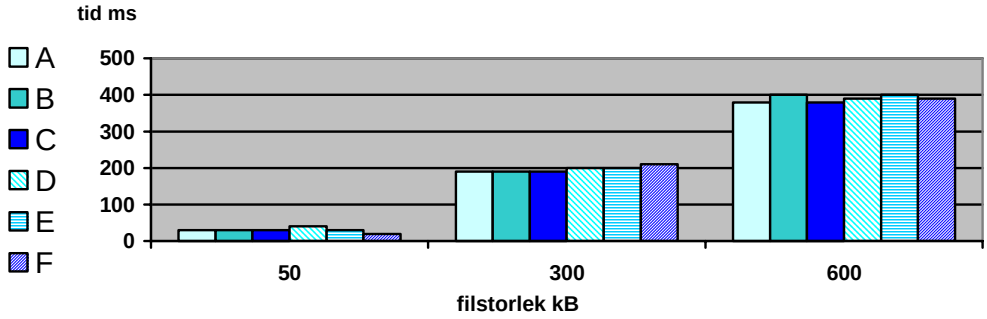
| | Antal preparerade träffar | Antal korrekta exempel som inte skall matcha | Tid |
|---|---|---|---|
| **A** | 100 | | **30** |
| **B** | 300 | | **30** |
| **C** | 600 | | **30** |
| **D** | 100 | 100 | **40** |
| **E** | 300 | 300 | **30** |
| **F** | 600 | 600 | **20** |

**300 kB filstorlek**

| | Antal preparerade träffar | Antal korrekta exempel som inte skall matcha | Tid |
|---|---|---|---|
| **A** | 100 | | **190** |
| **B** | 300 | | **190** |
| **C** | 600 | | **190** |
| **D** | 100 | 100 | **200** |
| **E** | 300 | 300 | **200** |
| **F** | 600 | 600 | **210** |

**600 kB filstorlek**

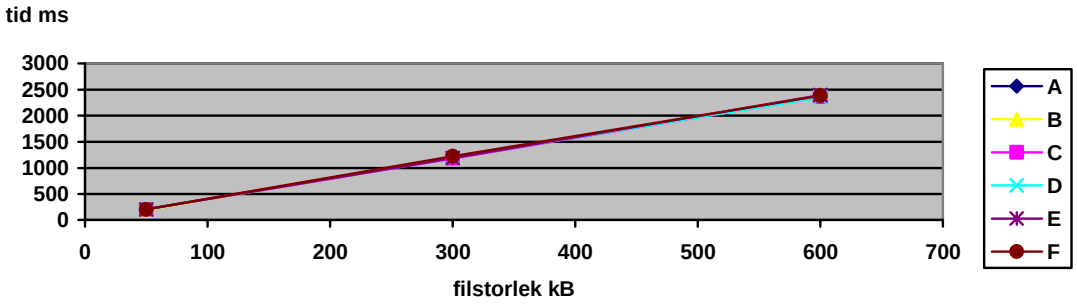
| | Antal preparerade träffar | Antal korrekta exempel som inte skall matcha | Tid |
|---|---|---|---|
| **A** | 100 | | **380** |
| **B** | 300 | | **400** |
| **C** | 600 | | **380** |
| **D** | 100 | 100 | **390** |
| **E** | 300 | 300 | **400** |
| **F** | 600 | 600 | **390** |



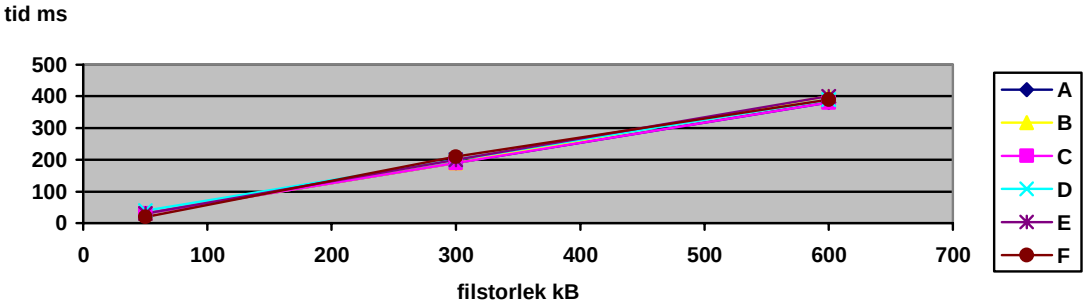

**Figur 8 DFA/NFA-lösningens resultat testat på genererade testfiler**

Antal träffar verkar inte göra någon större skillnad. Återigen finns det ett tydligt linjärt samband mellan filstorlek och tid vilket var väntat, se avsnitt 4.2 *Skillnader mellan NFA och DFA.*

Det linjära sambandet mellan tid och filstorlek illustreras väldigt väl i *Figur 9 och 10*.




**Figur 9 NFA-lösningens linearitet. Testfallen sammanfaller längs samma linje.**




**Figur 10 DFA/NFA-lösningens linearitet. Testfallen sammanfaller längs samma linje.**

## 7.6 Kompilatoroptimeringar

Hittills har inga särskilda kompilatordirektiv angetts. Utan optimeringsflaggor försöker kompilatorn, gcc, reducera den tid det tar att kompilera koden och att vid felsökning (debug) producera förväntat resultat. Satser är oberoende i den meningen att om man stoppar programmet vid en brytpunkt kan man tilldela variabler nya värden eller flytta programpekaren till en annan sats i definitionsområdet (se *man gcc* [12]).

Med optimerings flaggan ’–O1’ försöker kompilatorn reducera kodstorleken och exekveringstiden. Kompilatorn kommer också att lägga in vissa variabler i registerminnet för snabbare exekvering vilket annars bara sker om explicit angivet i koden.

NFA-lösningen tog 204303 ms utan optimering, med optimering tar det 203693 ms, en obetydlig skillnad. Det kan förklaras med att den kod som gör själva jobbet, regexbiblioteket, redan är optimalt kompilerad och testprogrammet länkar in denna kod.

För DFA/NFA tog lösningen 7350 ms med optimering jämfört med 10174 ms utan. Objektfilen för flexkoden minskade med 27kB från 70kB till 43 kB och huvudprogrammet minskade i motsvarande grad.

DFA/NFA-lösningen kan även optimeras genom att man ger flaggan –f till flex. Då kommer den minnesstruktur som representerar DFA inte att komprimeras. Objektfilen för flex-koden blir då mycket större, 434 kB istället för 70kB. Med optimeringsflaggan –f till flex tog DFA/NFA-lösningen 3725 ms.

Om man kombinerar de båda optimeringarna –O1 till gcc kompilatorn samt –f till flex tar DFA/NFA-lösningen 2143 ms. Objektfilen för flexkoden minskas återigen med 27 kB till 407 kB. Resultaten sammanfattas nedan:

**Optimeringsvinster för DFA/NFA-lösningen**

| | Exekveringstid i ms | Flex objektstorlek i kB |
|---|---|---|
| Ingen optimering | 10174 | 70 |
| gcc -O1 | 7350 | 43 |
| flex –f | 3725 | 434 |
| gcc –O1 samt flex –f | 2143 | 407 |

Det finns betydande vinster att göra med optimering påslagen. Att optimera flex objektkod med –f kostar dock en hel del minne.

# 8 Slutsatser

DFA/NFA-lösningen visar en förkrossande prestanda och är utan vidare den lösning som rekommenderas för Granska. Om bara några få regler skall undersökas går det förmodligen tillräckligt snabbt även för NFA-lösningen se avsnitt 7.4 *Prestanda som funktion av antal reguljära uttryck.*

Av resultaten i avsnitt 7.4 kan man även dra slutsatsen att oron för att lösningarna använder olika kod (se avsnitt 6.5 *Osäkerheter i mätningarna*) för filåtkomst inte kan påverka resultaten avsevärt mycket, ty i så fall skulle skillnaden inte vara så stor när NFA-lösningen använder mindre antal reguljära uttryck.

Dessutom kan man av resultaten i avsnitt 7.4 se att olika reguljära uttryck är betydligt jobbigare för båda lösningarna. Jämför resultaten då de första respektive sista 45 reglerna eftersöktes.

Ett tydligt linjärt samband mellan filstorlek och tid märks tydligt i avsnitt 7.5. Det talar också för DFA/NFA-lösningen då den långsamma NFA-delen endast behöver söka igenom träffarna och är oberoende av filstorleken. Givet att det är små texter (förkortningar e.dyl.) som först matchas av DFA-delen så innebär det att det är en väldigt liten textmassa NFA-delen behöver leta igenom. 200 träffar innebär en textmassa i storleksordningen 1 kB.

Det har tidigare framhållits att den kombinerade DFA/NFA-lösningen är komplex. Det är och kommer att vara jobbigare att underhålla denna lösning. I avsnitt 6.1 nämndes att det finns en syntaxskillnad mellan flex och GNU Regex. Skillnaden bestod i att mellanslag i flexfilens reguljära uttryck måste kapslas in mellan hakparenteser [ ].

Det är inte säkert att detta är den enda skillnaden mellan flex och GNU Regex. Det behövs mera testning för att täcka alla fall som reglerna avser testa.

# 9 Ordförklaringar

| | |
|---|---|
| DFA | Deterministisk finit automat, se avsnitt 4.2. |
| Flex | En fri version av Lex från Free Software Foundation. |
| Grep | Ett Unixverktyg för filtrering av text. |
| kB | Mått på filstorlek definierat som antal tusen tecken i en fil. |
| Lex | Ett programmeringsverktyg som använder sig av DFA. |
| Make | Ett regelstyrt programmeringsverktyg som används för bygge av komplexa program. |
| NFA | Icke-deterministisk finit automat, se avsnitt 4.2. |
| Perl | Ett populärt programmeringsverktyg vars funktionalitet för att hantera reguljära uttryck gjort språket mycket populärt. |
| POSIX | *Portable Operating System Interface*, en standard för portabla gränssnitt. |
| Sökmönstermotor | Programkod som kan tolka reguljära uttryck och matcha text mot dessa. |
| Unix | Ett vanligt förekommande operativsystem. |
| Word | Microsoft Word är ett vanligt ordbehandlingsprogram. |

# 10 Referenser


1. Aho. *Algorithms for Finding Patterns in Strings*, sid 255-300, Leeuwen (red.). *Handbook of Theoretical Computer Science Volume A, Algorithms and Complexity*. Elsevier Science Publishers 1990. ISBN: 0-444-88071-2

2. Domeij, Knutsson, Carlberger, Kann. *Granska – ett effektivt hybridsystem för kontroll av svensk grammatik*. NoDaLiDa, 49-56 dec 1999.

3. Friedl. *Mastering Regular Expressions*
   O'Reilly & Associates 1997. ISBN: 1565922573

4. Johansson. *NP-detektion – utvärdering och förslag till förbättringar av Granskas NP-regler.* C-uppsats i datorlingvistik, Institutionen för lingvistik, Stockholms Universitet 2000.

5. Klinting. *Castor bakar* Alfabeta 1996. ISBN 91 7712 720

6. Knuth, Morris, Pratt. *Fast pattern matching in strings*, SIAM Journal on Computing 6(1):323-350 1977.

7. Levine, Mason & Brown. *lex & yacc*. O'Reilly 1992. ISBN: 1-56592-0007

8. Ljung, Ohlander. *Allmän grammatik* Liber förlag 1985. ISBN: 91-38-60007-2

9. Parsons. *Compiler Construction* W. H. Freeman Company 1992. ISBN 0716782618

10. Paxson. *man flex.* Elektronisk manual som installeras på Unix operativsystem. Finns fritt tillgängligt på internet via sökorden ”man flex”.

11. Spencer. *man regex.* Elektronisk manual som installeras på Unix operativsystem. Finns fritt tillgängligt på internet via sökorden ”man regex”.

12. Stallman. *man gcc.* Elektronisk manual som installeras på Unix operativsystem. Finns fritt tillgängligt på internet via sökorden ”man gcc”.

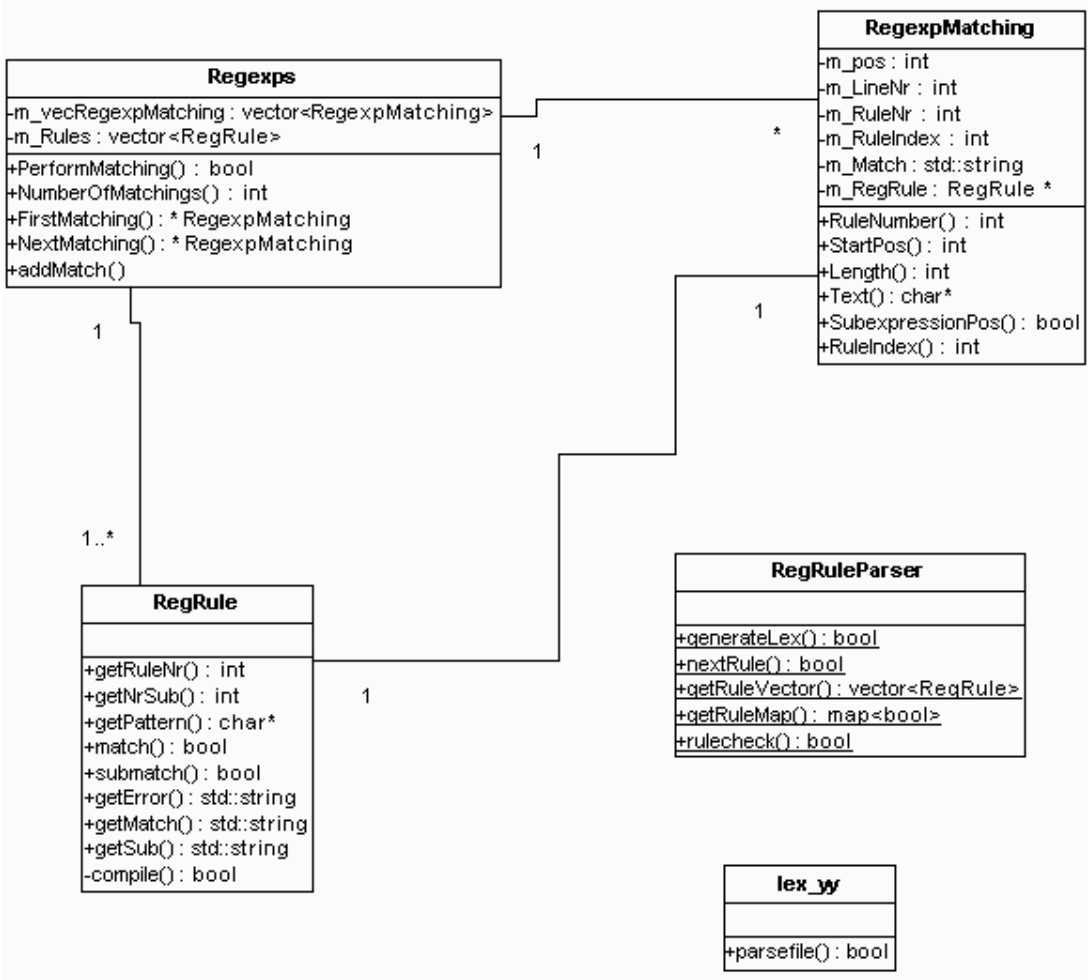
# Bilaga - Programdokumentation

Viss kunskap om den kombinerade DFA/NFA-lösningen krävs när man skall underhålla och felsöka i koden. Nedan följer en kort illustrerad genomgång av de viktigaste klasserna. Klassernas användning beskrivs sedan i tre scenarion. En fullständig listning av publika metoder bifogas sist i avsnittet.

**Klassdiagram**
Gränssnittet mot Granska består av två klasser: Regexps och RegexpMatching. Regexps tillhandahåller den funktionalitet som behövs och RegexpMatching data om hittad träff.

Internt finns en klass (RegRule) som innehåller alla regler inlästa från regelfilen. RegRule har funktionalitet för att matcha en given sträng med hjälp av GNU regex och ge bakåtreferenser. Klassen RegRuleParser har den funktionalitet som behövs för att den skall kunna gå igenom regelfilen och generera en lexfil eller läsa in alla regler till en vektor.

Klassernas inbördes relationer visas i nedanstående klassdiagram. Klassen Regexps har en lista av alla regler (RegRule) samt en lista över alla träffar (RegexpMatching) som påträffats.

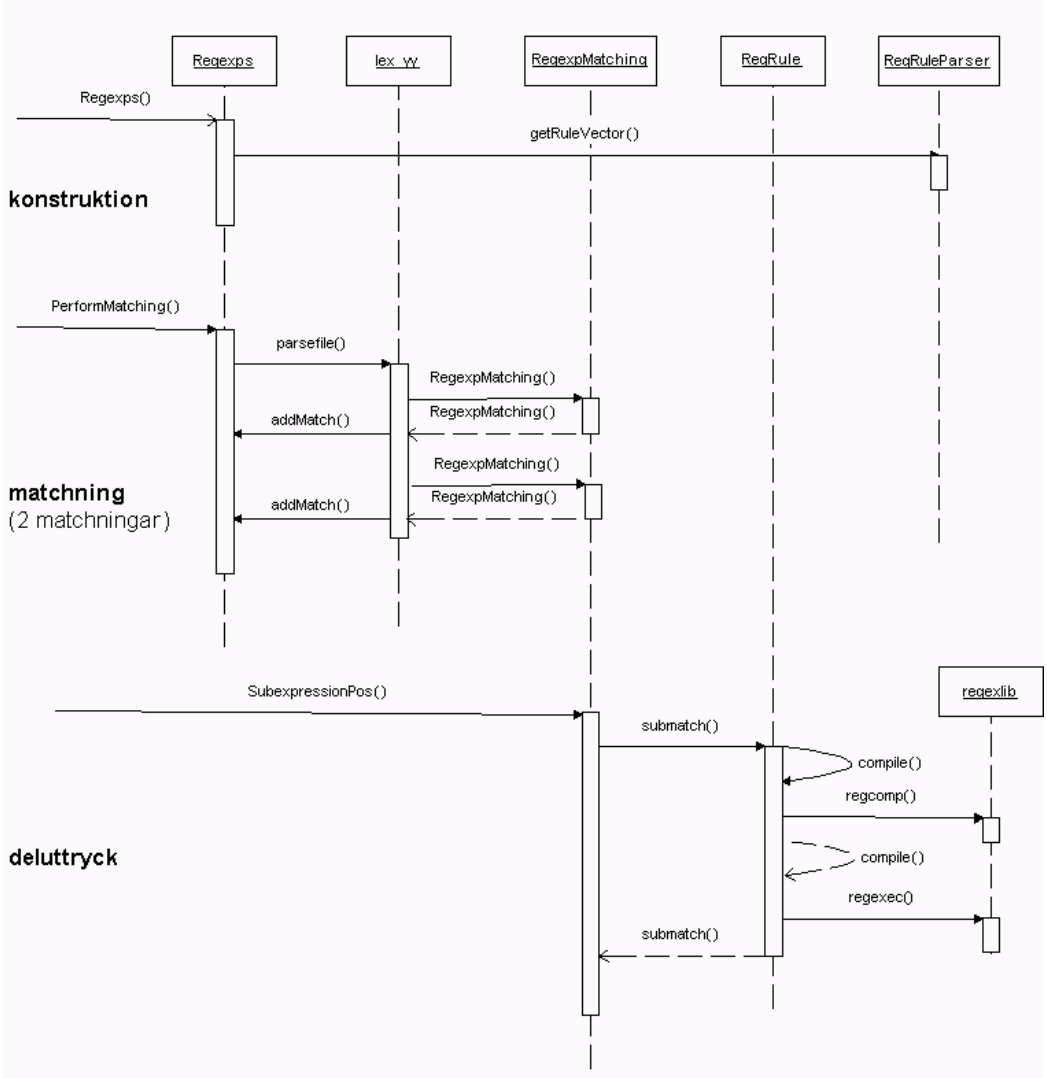


**Scenario - Konstruktion**
När Regexps konstrueras anropas RegRuleParser::getRuleVector som läser in alla regler i regelfilen till en vektor i Regexps.

**Scenario - Matchning**
Då sökning påbörjas anropas en funktion definerad i lex. För varje träff bildas en RegexpMatching. Träffen kopieras omedelbart in i en vektor i Regexps tillsammans med data om vart den påträffades i filen.

**Scenario - Deluttryck**
När bakåtreferenser eller deluttryck efterfrågas anropas en funktion i RegRule med en textsträng som argument. Om det är första gången denna regel efterfrågas kompileras regelns sökmönster med GNU Regex (NFA). Därefter används GNU Regex för matchning av den givna textsträngen.

Ovanstående scenarion visas i nedanstående diagram.



## Class Regexps

```
//////////////////////////////////////////////////////////////////////
//
// PerformMatching      applies lex on file
//
// RETURNS
//  bool
// PARAMETERS
//  FILE * file         text file
//
//////////////////////////////////////////////////////////////////////
//
// NumberOfMatchings    returns number of matches
//
// RETURNS
//  int
//
//////////////////////////////////////////////////////////////////////
//
// FirstMatching        returns pointer to first match
//
// RETURNS
//  RegexpMatching *
//
//////////////////////////////////////////////////////////////////////
//
// NextMatching         returns pointer to next match
//
// RETURNS
//  RegexpMatching *
// PARAMETERS
//  RegexpMatching *r   the match from which to return the following
//                      match
//
//////////////////////////////////////////////////////////////////////
//
// addMatch             called by lex to add match
//
// RETURNS
//  void
// PARAMETERS
//  int pos             position of match in text file
//  const char * match  the match
//  int ruleindex       ruleindex in rulevector
//  int rulenr          rulenumber in rulefile
//  int linenr          line in text file where match was found
//
```

## Class RegexpMatching

```
//////////////////////////////////////////////////////////////////////
//
// StartPos             Returns start position in file of match
//
// RETURNS
//  int                 start position in file (OBS, \r in DOS files
//                      counted)
//
//////////////////////////////////////////////////////////////////////
//
// Length               returns length of match
//
// RETURNS
//  int
//
//////////////////////////////////////////////////////////////////////
//
// Text                 returns the match
//
// RETURNS
//  const char *
//
//////////////////////////////////////////////////////////////////////
//
// RuleIndex            index to the rule that matched
//
// RETURNS
//  int                 index in Regexps rulevector
//
//////////////////////////////////////////////////////////////////////
//
// RuleNumber           returns the rule that matched
//
// RETURNS
//  int                 rule number in rulefile
//
//////////////////////////////////////////////////////////////////////
//
// MatchIndex           number of matches when this match was found
//
// RETURNS
//  int                 index in Regexps matchvector
//
//////////////////////////////////////////////////////////////////////
//
// SubexpressionPos     returns position of subexpression
//
// RETURNS
//  bool                false if out of bounds, regex compilation
//                      failed ...
// PARAMETERS
//  int i               nr of subexpression
//  int &start          start is relative the matching
//  int &length         length of subexpression
//
```

## Class RegRule

```
//////////////////////////////////////////////////////////////////////
//
// getPattern           returns the regular expression for this rule
//
// RETURNS
//  const char *
//
//////////////////////////////////////////////////////////////////////
//
// getRuleNr            returns rule number as found in rulefile
//
// RETURNS
//  int                 rule number in rulefile
//
//////////////////////////////////////////////////////////////////////
//
// getNrSub             returns the max nested number of
//                      subexpression as reported by regex lib when
//                      compiled
//
// RETURNS
//  int                 max nested number of subexpression
//
//////////////////////////////////////////////////////////////////////
//
// classinit            Put default values of on all members,
//
// RETURNS
//  void
//
//////////////////////////////////////////////////////////////////////
//
// compile              Compiles the regexp pattern with gnu_regexp
//                      package, sets m_bCompiled to true if
//                      succesful. If compile() fails it sets m_Error
//                      and returns false m_bCompiled does not change
//
// RETURNS
//  bool                false if regex compilation failed.
//
//////////////////////////////////////////////////////////////////////
//
// submatch             returns matched subexpression
//
// RETURNS
//  bool
// PARAMETERS
//  const string line   string to apply regex on
//  int subnr           nr of subexpression, 0 is whole expression.
//  int &start          starting char in line of subexpression
//  int &length         length of subexpression
//
```

```
//////////////////////////////////////////////////////////////////////
//
// match                deprecated, see submatch above
//
// RETURNS
//  bool
// PARAMETERS
//  const string line
//
//////////////////////////////////////////////////////////////////////
//
// getError             returns error status
//
// RETURNS
//  string              empty string means no error
//
```

## Class RegRuleParser

```
//////////////////////////////////////////////////////////////////////
//
// rulecheck            Validates regex rules by parsing rulefile and
//                      applying match/nonmatch examples.
// RETURNS
//  bool                true if all rules passed
// PARAMETERS
//  string filename     path to rulefile
//  ostream & os        output
//  bool showAll        if false, only failing rules are output
//
//////////////////////////////////////////////////////////////////////
//
// getRuleVector        parses rulefile, creates and returns vector
//                      of rules
//
// RETURNS
//  vector<RegRule>
// PARAMETERS
//  string filename     path to rulefile
//
//////////////////////////////////////////////////////////////////////
//
// nextRule             advances filepointer to beginning of next
//                      rule
// RETURNS
//  bool                returns false when EOF reached
// PARAMETERS
//  FileHandler & file  rulefile FileHandler
//
//////////////////////////////////////////////////////////////////////
//
// generateLex          see generateLex below
//
// RETURNS
//  bool
// PARAMETERS
//  string extr_file    path to rulefile
//  ostream & os
//
```

```
//////////////////////////////////////////////////////////////////////
//
// generateLex          parses rulefile and generates lex file
//
// RETURNS
//  bool                always true
// PARAMETERS
//  FileHandler & file  rulefile FileHandler
//  ostream & os        output of lex file
//
//////////////////////////////////////////////////////////////////////
//
// modifyRegexToLex     modifies regular expression to lex
//                      specific syntax: space -> [space]
// RETURNS
//  bool
// PARAMETERS
//  std::string & reg   regular expression to modify
//  int iRuleNr         for trace output only
//
//////////////////////////////////////////////////////////////////////
//
// isInsideBrackets    checks if character at pos is inside []
//
// RETURNS
//  bool
// PARAMETERS
//  string Pattern
//  int    pos          position to search from
//
//////////////////////////////////////////////////////////////////////
//
// RuleNrToInt         Parses line in rulefile and extracts
//                     first integer found as rulenr.
// RETURNS
//  int
// PARAMETERS
//  string RuleNr       line in rulefile
//
```

## Exempelkod på användning av Regexps

```
  regexps.PerformMatching(file);

  RegexpMatching * regexpmatching = regexps.FirstMatching();
  while (regexpmatching != NULL) {
    string text = regexpmatching -> Text();
    int start, length;
    regexpmatching -> SubexpressionPos(i, start, length);

    regexpmatching = regexps.NextMatching(regexpmatching);
  }
```